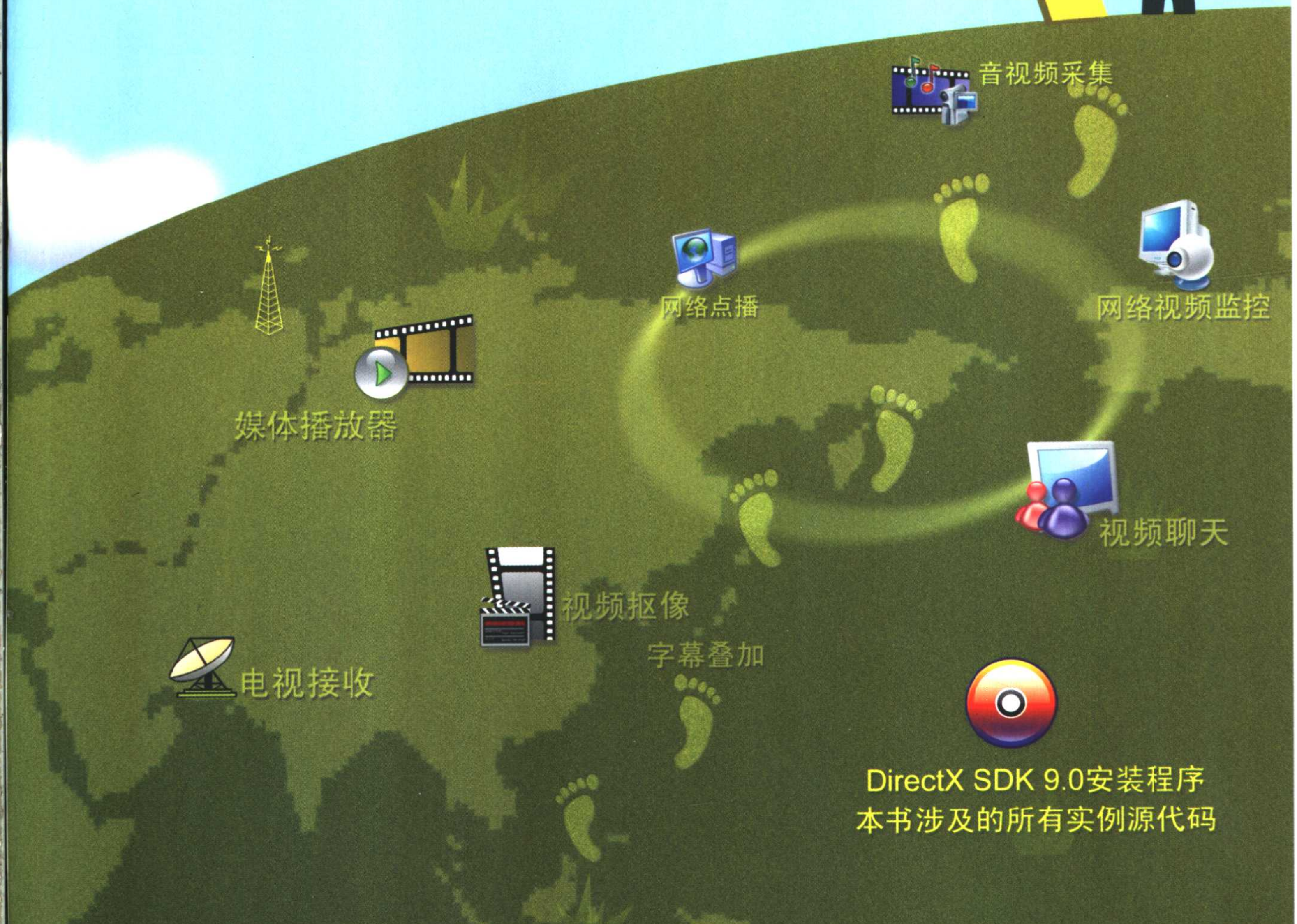
DirectShow 专业著作

DirectShow 实务精选

陆其明 著



科学出版社　北京科海电子出版社

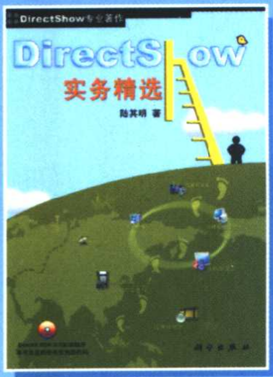
DirectShow系列丛书

《DirectShow开发指南》

一本全面介绍DirectShow技术框架及其应用的书籍。全书共分4个部分：第1部分详细介绍了DirectShow的基础知识；第2部分重点讨论了Filter的开发，以及DirectShow应用程序的开发；第3部分深入分析了DirectShow SDK提供的部分典型源代码；第4部分结合作者个人的一些开发实践，通过案例和开放源码分析，进一步介绍了DirectShow的实务应用。

该书内容丰富，条理清晰，实用性强。适合广大的流媒体应用开发人员，以及对Windows平台上多媒体处理感兴趣的编程爱好者学习和参考。

《DirectShow实务精选》



一本侧重介绍DirectShow技术在当前主要几个领域中应用的书籍。这些应用包括音视频采集、视频聊天、视频点播、视频叠加、媒体播放等。全书结构合理，内容丰富新颖、条理清晰。在介绍各种应用的同时，辅助以大量详尽的实例源代码。这些实例，不仅具有很强的实用价值和指导意义，更让读者理论联系实践，使学习DirectShow技术变得轻松自如。

该书适合广大的流媒体应用开发人员、系统设计人员、以及对Windows平台上多媒体处理感兴趣的编程爱好者学习和参考。

ISBN 7-03-013654-3
9 787030 136541 >
定价：59.00元（含光盘）

封面设计：谭洁红
读者联系电话：（010） 82896448，82896450
网　址：http://www.khp.com.cn
E-mail：feedback@khp.com.cn

DirectShow 实务精选

陆其明　编著

科学出版社
北京科海电子出版社

内 容 提 要

本书是"DirectShow 专业著作"丛书的第二本，侧重介绍了 DirectShow 技术在当前主要的几个领域中的应用，包括音视频采集、视频聊天、视频点播、视频叠加、媒体播放等。在介绍各种应用的同时，辅助以大量详尽的实例源代码。这些实例，不仅具有很强的实用价值和指导意义，更让读者理论联系实践，使学习 DirectShow 技术变得轻松自如。

本书结构合理，内容丰富新颖、条理清晰，适合广大的流媒体应用开发人员、系统设计人员、以及对 Windows 平台上多媒体处理感兴趣的编程爱好者学习和参考。

图书在版编目（CIP）数据

DirectShow 实务精选/陆其明编著.—北京：
科学出版社，2004.7
ISBN 7-03-013654-3

I. D… II. 陆… III. 多媒体—软件工具，Direct Show
IV. TP311.56

中国版本图书馆 CIP 数据核字（2004）第 054879 号

责任编辑：洪英　/ 责任校对：科海
责任印刷：科海　/ 封面设计：谭洁红

科 学 出 版 社 出版
北京东黄城根北街 16 号
邮政编码：100717
http://www.sciencep.com
北京科普瑞印刷有限责任公司印刷
科学出版社发行　各地新华书店经销

*

2004 年 7 月第一版　开本：16 开
2004 年 7 月第一次印刷　印张：30
印数：1~4 000　字数：730 千字

定价：59.00 元（1CD）
（如有印装质量问题，我社负责调换）

前　言

笔者的前一本拙作《DirectShow开发指南》（清华大学出版社，2003年）自问世以来，颇受业内人士的关注。有褒扬的、有批评的，更多的则是对笔者提出了新的建议。首先，笔者要衷心地感谢这些热心的朋友，正是这些热心读者的关心和支持，使笔者深受鼓舞，才决定要再编写一本书。

《DirectShow开发指南》比较详尽地介绍了DirectShow系统框架，以及DirectShow技术应用的方方面面。但是，有些读者仍然反馈道："书的内容讲得细致入微，我都看懂了。但在实际工作中，碰到问题后还是不知道从何处下手解决"，"看了这本书，基本上学会了怎么开发Filter，但是在应用程序中使用自己开发的Filter时，碰到了很多问题……"，"这本书比较偏重于Filter的开发，能不能再写一本书，专门介绍DirectShow技术的实际应用呢？"看到这些读者的反馈后，笔者心里十分感激，笔者首先要对他们说的是：实践出真知，要勇于实践。学习一门新技术，贵在理论联系实践。只有把从书本上学到的东西放到实践中去检验，才能加深对已学知识的理解。反复实践，善于在实践中思考、总结，对于提高自身的技术水平，以及分析问题、解决问题的能力是至关重要的。之后，笔者也对《DirectShow开发指南》一书的内容进行了一番反思：在这方面，笔者是不是也有做得不够的地方呢？或许，一些读者已经学会了如何制砖，但是，他们还不知道如何使用这些砖头来造房子。考虑许久之后，笔者决定编写本书（专门介绍如何造房子，造各式各样的房子）。

本着"实用第一"的原则，笔者开始写作本书。为了避免与前一本书在内容上的重叠，本书侧重介绍了DirectShow技术在当前主要的几个领域中的应用，包括音视频采集、视频聊天、视频点播、视频叠加、媒体播放等。在介绍各种应用的同时，辅助以大量详尽的实例源代码。这些实例，都是经过笔者精心设计的，不仅具有很强的实用价值和指导意义，更能促使读者理论联系实践，真正地提高对DirectShow技术的实际应用能力。读完本书之后，读者如果能够从容面对当前市面上绝大部分的DirectShow相关的应用开发，那么，这本书的价值也就得到了体现。

本书是集体智慧的结晶。衷心感谢出版社的夏非彼老师、编辑李才应等人，如果没有这些朋友的帮助，就不会有这本书的问世。本书在写作过程中，得到了敏递软件（上海）有限公司的总经理祝开景博士的关心和支持，还有金邦飞、卞劲松、马涛、季劲松等人的帮助，衷心感谢他们："你们是我碰到过的最好的同事！"本书的封面照旧由我的爱人谭洁红设计，书中各章的插图也出自于她的创意。正是她出色的工作，使本书增色不少！由于时间短，任务重，再加之笔者的水平有限，书中的错误以及疏漏之处在所难免，望各位

专家、同行批评指正。

注：本书的配套光盘提供了书中各章节涉及的所有实例源代码。这些实例均在Windows 2000操作系统、DirectX SDK 9.0、VC 6.0开发环境下编译、测试通过。

声明：本书提供的实例源代码仅限于学习参考，若用于商业用途，本人不对源代码的安全性和稳定性、以及因此可能导致的任何损失负责。请读者自行斟酌。

陆其明

2004年6月于上海

目　录

第1章　DirectShow系统 1

1.1　系统概述 1
1.2　使用GraphEdit 3
1.3　最后的话 6

第2章　音视频采集 9

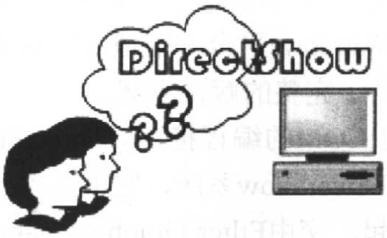
2.1　功能介绍 9
2.2　通用采集 13
2.2.1　采集设备枚举 13
2.2.2　采集设备创建 25
2.2.3　开始视频预览 29
2.2.4　使用高效的VMR 45
2.2.5　选择输入端子 55
2.2.6　采集参数设置 64
2.2.7　支持数码摄像机 76
2.2.8　支持热插拔 82
2.2.9　支持即时抓图 93
2.2.10　采集到AVI文件 114
2.2.11　视频的压缩方案 118
2.3　电视接收 121
2.3.1　预备工作 121
2.3.2　电视选台 124
2.3.3　支持VBI 125
2.4　程序导读 130
2.4.1　一种不错的设计模式 130
2.4.2　程序结构 137
2.4.3　流程概要 141

第3章　网络应用 144

3.1　网络编程基础 144
3.1.1　TCP传输 145
3.1.2　UDP传输 166
3.1.3　IP组播技术 176

3.2 视频聊天 185
3.2.1 功能介绍 185
3.2.2 实现原理 187
3.2.3 采集设备自检 190
3.2.4 角色控制实现 204
3.2.5 开发网络发送Filter 223
3.2.6 开发网络接收Filter 230
3.2.7 程序导读 240
3.3 视频点播 244
3.3.1 功能介绍 244
3.3.2 实现原理 247
3.3.3 开发网络接收Filter 250
3.3.4 支持MPEG1、MP3 261
3.3.5 支持MPEG2 268
3.3.6 支持AVI 270
3.3.7 程序导读 275
第4章 视频叠加 311
4.1 视频抠像 311
4.1.1 抠像原理 311
4.1.2 抠像算法实现 312
4.1.3 MMX算法优化 318
4.1.4 开发抠像Filter 321
4.1.5 模拟实时源 339
4.1.6 应用演示 348
4.2 字符叠加 350
4.2.1 开发字符叠加Filter 350
4.2.2 与媒体播放集成 353
第5章 SDK源码赏析 361
5.1 DSNetwork例子 361
5.1.1 程序结构分析 361
5.1.2 模拟MPEG2-TS源 372
5.1.3 应用演示 380
5.2 VMR-9典型应用 385
5.2.1 位图叠加例子BitmapMix 385
5.2.2 字符叠加例子Ticker 396
5.2.3 流混合例子Blender 405

附录A　DirectShow常见问题解答 411
A.1　一般性问题 411
A.2　编程问题 413
附录B　PID扩展插件开发 420
B.1　PID应用原理 420
B.2　PID插件开发 421
B.3　PID插件的使用 429
附录C　DES扩展Source（实现字符叠加） 432
C.1　自动化基础 433
C.2　字符叠加基础 433
C.3　Source Filter的开发 436
C.4　Source Filter的调试 448
C.5　DES应用举例 449
附录D　让Windows Media Player播放自定义格式文件 452
D.1　播放媒体文件的Filter Graph构建过程 452
D.2　两种解决方案 454
D.3　自定义格式文件的生成 455
D.4　Source Filter的开发 457
D.5　体验QQ文件的播放 471



1 DirectShow 系统

1.1 系统概述

很多人都听说过DirectX，而且知道它神通广大，因为很多软件（特别是一些游戏）都钦点了它——要求系统安装DirectX xx以上版本。那么，DirectX到底是什么呢？它有哪些特殊的本领呢？其实，DirectX是微软公司开发的一套基于Windows平台的编程接口（API）；它能出色地完成高速的实时动画渲染、交互式音乐和环境音效、高效多媒体数据处理等一般API很难完成的任务。

从Windows 95开始，几乎每一代Windows操作系统都集成了一定版本的DirectX运行时库（Runtime Library），它们的版本对应关系如表1.1所示。

表1.1　Windows操作系统与DirectX的版本对应

操作系统版本	DirectX版本
Windows 95	2.0
Windows NT 4.0	3.0a
Windows 98	5.0
Windows 2000	6.0
Windows Me	7.0
Windows XP	8.1

事实上，DirectX已经成为Windows家族操作系统中不可或缺的核心组件之一。这些组

件可以不断地升级到最新版本，但一般不可以卸载。目前，DirectX的最新版本是9.0，它的安装程序可到微软公司的官方网站http://www.micorsoft.com/directx上免费下载（本书的配套光盘也提供）。

DirectX的家族成员很多，而且各有各的本领，如DirectDraw和Direct3D负责二维图形图像/三维动画加速、DirectMusic和DirectSound负责交互式音乐/环境音效处理等。DirectShow是DirectX大家族中的一位成员，DirectShow为Windows平台上处理各种格式的媒体文件播放、音视频采集等高性能要求的多媒体应用提供了完整的解决方案。

DirectShow是一个开放性的应用框架，也是一套基于COM的编程接口。DirectShow的系统功能如图1.1所示。可以看到，图中最大的一块即是DirectShow系统，它的基本工作原理就是“流水线”：将单元组件——Filter——串联在一起，交由Filter Graph Manager统一控制。系统的输入可以是本地文件系统、硬件插卡、因特网等，系统的输出可以是声卡（声音再现）、显卡（视频内容显示）、本地文件系统，当然也可以最终将数据向网络发送。

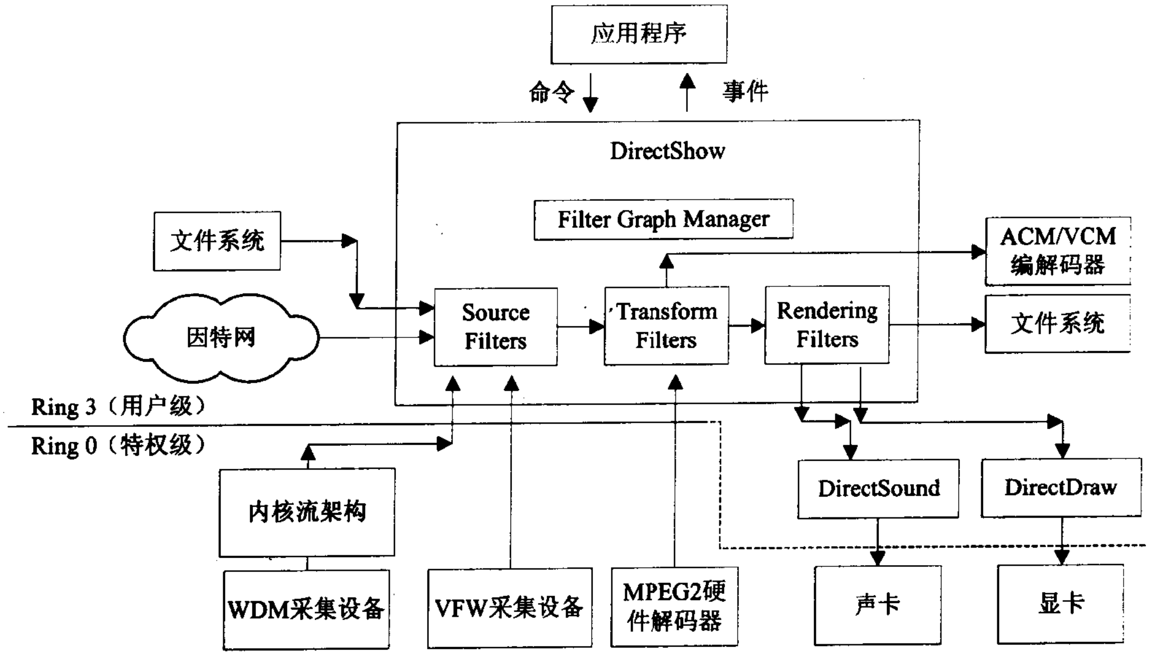


图 1.1 DirectShow 系统功能

事实上，计算机应用领域中的很多模块都可以和DirectShow系统交互。也就是说，DirectShow的应用范畴很广。单纯从本地系统来说，DirectShow可以实现不同格式的媒体文件的解码播放或格式之间的相互转换，可以从本地机器中的采集设备采集音视频数据并保存为文件，可以接收、观看模拟电视等。而从网络应用的角度来说，DirectShow更可用于视频点播、视频会议、视频监控等领域。其实，从广义上来说，DirectShow系统适合于一切流式数据的处理，这些数据可以是音频、视频这样的多媒体数据，但又不局限于多媒体数据。

更多的感性认识DirectShow可以借助于GraphEdit——一个随DirectX SDK一起发布的小型工具软件（即SDK目录中的Bin\DXUtils\graphedt.exe）。下面，就来介绍这个工具软件的使用。

1.2　使用GraphEdit

GraphEdit可以算是一个简单的DirectShow应用程序。它有标准的Windows用户界面，如图1.2所示。

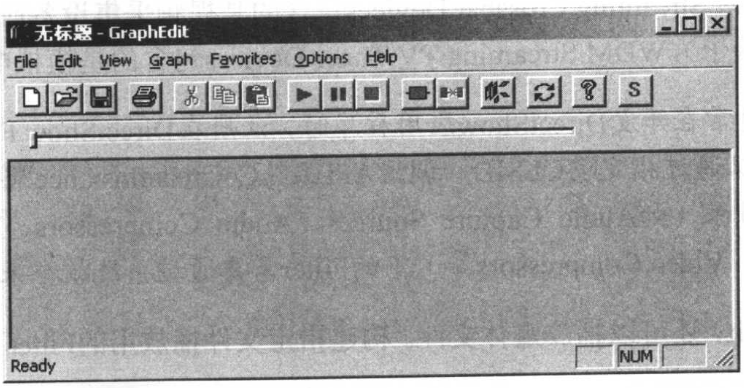


图 1.2　GraphEdit 程序界面

使用GraphEdit可以查看到系统中所有（正常）安装的Filter。执行菜单命令Graph | Insert Filters，将弹出一个如图1.3所示的对话框。

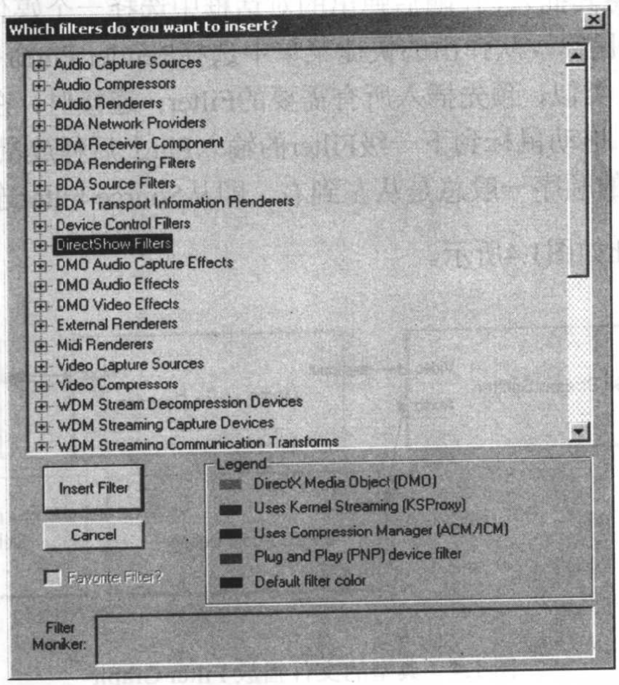


图 1.3　使用 GraphEdit 查看系统中的 Filter

提示：Filter的注册信息通常包括两部分——COM服务器信息和Filter描述信息。前者是必需的，它表明了Filter是一个COM组件；后者不是必需的，不注册也不会妨碍到Filter功能的实现。GraphEdit查看不到没有注册Filter描述信息的Filter。

图1.3中有很多Filter目录，其中最主要的是DirectShow Filters（图1.3中高亮显示），因为绝大部分的Filter都注册在这个目录下。其他的目录，如Audio Capture Sources包含的是音频采集设备，Audio Compressors包含的是音频各种格式的编码Filter，Video Capture Sources包含的是视频采集设备，Video Compressors包含的是视频各种格式的编码Filter，WDM Streaming Capture Devices包含的是本地机器中所有的采集设备（包括音频采集设备和视频采集设备），WDM Streaming Crossbar Devices包含的是视频采集设备前端的路由装置（用于选择视频输入端子），WDM Streaming TV Tuner Devices包含的是模拟电视接收设备等等。

提示：通常在开发DirectShow应用程序时，注册在DirectShow Filters目录下的Filter可以通过指定其CLSID、调用API函数CoCreateInstance来创建，而注册在其他目录（如Audio Capture Sources、Audio Compressors、Video Capture Sources、Video Compressors等）下的Filter需要通过系统枚举来创建。

使用GraphEdit，还可以播放媒体文件。构建指定文件播放用的Filter Graph至少有以下4种方法：

（1）执行菜单命令File | Render Media File，在随后弹出的对话框中选择一个媒体文件。

（2）在文件浏览器中直接将要播放的媒体文件拖到GraphEdit中。

（3）执行菜单命令Graph | Insert Filters，在DirectShow Filters目录下找到File Source（Async.），双击鼠标将其插入，在随后弹出的对话框中选择一个媒体文件，然后再在Source Filter的输出Pin上右击鼠标，从弹出的快捷菜单中选择Render Pin命令。

（4）与方法（3）类似，预先插入所有需要的Filter；连接时，在上一级Filter的输出Pin上按住鼠标左键不放，拖动鼠标到下一级Filter的输入Pin上后释放鼠标；这样依次连接所有必要的Pin即可（连接的顺序一般总是从左到右，即从Source Filter连到Renderer Filter）。

构建的Filter Graph如图1.4所示。

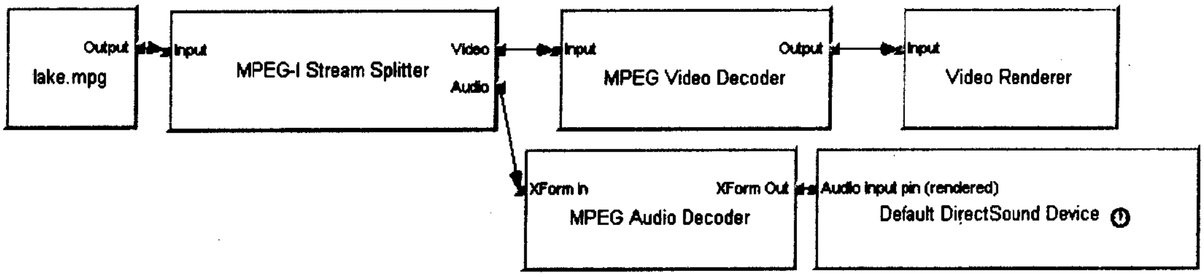


图 1.4 典型的文件播放 Filter Graph

当Filter Graph构建完成后就可以进行播放了。执行菜单命令Graph | Play、Pause或Stop，或者直接使用工具条上的播放控制按钮。

使用GraphEdit，还可以帮助我们调试自己开发的Filter或者DirectShow应用程序。在Filter的开发项目上，设置VC项目DEBUG时的执行程序为graphedt.exe。当GraphEdit运行后，插入我们的Filter，构建特定的Filter Graph，然后运行Filter Graph就可以跟踪Filter的实际执行流程了。而在开发DirectShow应用程序时，最通常的做法也是先在GraphEdit中搭建特定

的Filter Graph进行测试。效果满意了，然后再以GraphEdit中的Filter连接图为模板，在应用程序中用程序代码来构建。

在应用程序中构建的Filter Graph不像在GraphEdit中构建的那么直观。要是在应用程序中构建的Filter Graph能用GraphEdit显示出来就好了！幸运的是，GraphEdit提供了这样的功能，方法是：执行菜单命令File | Connect to Remote Graph，随后将弹出如图1.5所示的对话框，选中一个Filter Graph的注册条目后单击OK按钮即可。

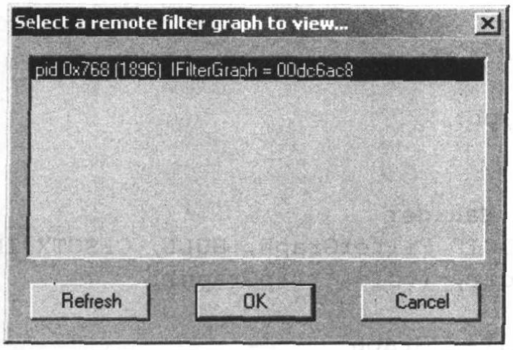


图 1.5 GraphEdit 的远程连接功能

需要注意的是，GraphEdit的这个远程连接功能仅在Windows 2000和Windows XP操作系统下能用，而且还有一个前提是，应用程序必须把它创建的Filter Graph进行注册，如通过调用AddToRot函数实现：

```
// 在应用程序成功创建了一个Filter Graph之后调用AddToRot进行注册
// pUnkGraph：Filter Graph Manager对象指针
// pdwRegister：成功注册后返回的一个ID
HRESULT AddToRot(IUnknown *pUnkGraph, DWORD *pdwRegister)
{
    IMoniker * pMoniker;
    IRunningObjectTable *pROT;
    if (FAILED(GetRunningObjectTable(0, &pROT)))
    {
        return E_FAIL;
    }
    WCHAR wsz[256];
    wsprintfW(wsz, L"pid %08x IFilterGraph = %08x ",
        GetCurrentProcessId(), (DWORD_PTR)pUnkGraph);
    HRESULT hr = CreateItemMoniker(L"!", wsz, &pMoniker);
    if (SUCCEEDED(hr))
    {
        hr = pROT->Register(ROTFLAGS_REGISTRATIONKEEPSALIVE,
            pUnkGraph, pMoniker, pdwRegister);
        pMoniker->Release();
    }
    pROT->Release();
    return hr;
}

// 在销毁Filter Graph Manager对象之前，调用RemoveFromRot进行反注册
```

```
// pdwRegister为成功调用AddToRot函数后返回的ID
void RemoveFromRot(DWORD pdwRegister)
{
    IRunningObjectTable *pROT;
    if (SUCCEEDED(GetRunningObjectTable(0, &pROT)))
    {
        pROT->Revoke(pdwRegister);
        pROT->Release();
    }
}
```

应用程序中的整个操作过程如下：

```
IGraphBuilder *pGraph;
DWORD dwRegister;

// 创建Filter Graph Manager
CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER,
    IID_IGraphBuilder, (void **)&pGraph);

// 仅在Debug时注册Filter Graph
#ifdef _DEBUG
hr = AddToRot(pGraph, &dwRegister);
#endif

// 应用程序进行其他操作（省略）
// ……

// 对Filter Graph进行反注册
#ifdef _DEBUG
RemoveFromRot(dwRegister);
#endif

// 销毁Filter Graph Manager对象
pGraph->Release();
```

提示：掌握GraphEdit更多的使用方法，请读者参考它的帮助文档（即SDK目录中的Bin\DXUtils\GraphEdit.chm）。

1.3 最后的话

开发DirectShow应用程序，应该选择哪种编程语言？笔者推荐使用VC。因为DirectX SDK最初就是为VC开发者设计的。要想深入理解DirectShow架构，没有一定的VC编程基础也是不行的。那是不是说，VB、DELPHI等其他语言的开发者就不能使用DirectShow了呢？非也！DirectShow应用程序其实就是一种COM客户程序，开发语言只要支持COM组件的使用就能使用DirectShow。

提示：对于VB、DELPHI等其他语言的开发者来说，除了直接使用DirectShow SDK外，还可以有另外一种方法：使用VC对DirectShow做一层封装，生成一

个DLL，应用程序上直接调用这个DLL，间接使用DirectShow提供的功能服务。

既然说到了编程语言，下面就顺便介绍一下笔者的编程风格。因为从第2章开始，读者将接触到大量笔者编写的演示程序。预先了解一下笔者的编程风格，对于理解这些程序将会很有帮助。

笔者喜欢面向对象的程序设计风格，因此会写很多类。类有很多好处，如封装性、继承性、多态性，更接近于人的自然思维等。类的提取很大程度上取决于对专业应用领域的了解，以及个人的程序设计经验。类的功能定义应该是很明确的，如这个类主要做什么，哪些属性是私有的、哪些是保护的（通常情况下不要定义公有属性），哪些接口要对外暴露、哪些应该隐藏等。很多事物，规模（数量）大了就会凸现一个管理的问题，类也是这样。笔者一般在一个文件中只定义一个类，而且文件的名字跟类的名字一致（VC默认情况下，文件名会把类的前缀C去掉）。例如现在有一个Filter Graph Manager的封装类CDXGraph，它的实现文件就是CDXGraph.cpp和CDXGraph.h。而且，笔者喜欢把同一个类的.cpp文件和.h文件放在一起（VC默认情况下是分开的，.cpp文件放在Source Files目录下，.h文件放在Header Files目录下）。根据类完成的功能，在VC的工作区（Workspace）建立不同的目录，以分别对这些类进行管理。这么做对于一个稍大的项目来说尤为重要。不要一时之间编写了很多类，回过头来想要查找某个类时却要费九牛二虎之力。如此就麻烦了！

笔者喜欢界面、控制逻辑、数据分离的三层结构的程序设计方法。在笔者提供的演示实例中，很多都是基于MFC对话框类型的程序，界面类就是标准的对话框类CDialog的派生类。一般在界面类中会定义几个自定义类的实例，这些自定义类，典型的情况下，有一个是数据封装类，一个是逻辑控制类。这些类的作用是：界面类表达一种用户操作逻辑；数据封装类集中管理各种参数；逻辑控制类具体实现业务应用逻辑，类名一般类似于CXXXController、CXXXManager、CXXXAdministrator等。这种界面、控制逻辑、数据分离的三层结构，不仅使程序框架清晰、容易被人理解，而且具有良好的移植性和扩展性。

笔者不喜欢在代码中写很多注释，除非特别需要（如一些核心算法、异常情况等）。如果一定要给一段代码写上大段的注释后才能让人理解，笔者首先会怀疑这段代码的合理性。笔者认为，通过合适的函数命名以及参数、变量的命名，足以让程序具有很好的可读性。在类中定义成员变量，变量名中一般不带下划线（太麻烦了），如定义一个long型变量，VC推荐的是long m_Length，而笔者更喜欢写成long mLength。对于一些临时变量，笔者一般不会把变量的类型写到变量名中去，如定义int nFileSize，笔者更喜欢写成int fileSize。因为如果以后想改变变量的类型为long，就没有必要同时改变变量名。（如果像原先定义的int nFileSize，变量类型改成long之后，为了一致起见，变量名应该相应改成lFileSize；另外，程序中所有用到该变量的地方都要一一改正。太麻烦！）变量名要取得有实际意义，如果由几个单词组成，第一个单词全部小写，后面的单词仅第一个字母大写。对于函数的参数，参数名也不带类型信息，而是用前缀in表示输入参数，out表示输出参数，io表示输入并有输出的参数。例如函数BOOL Transform(BYTE * inData, long * ioLength, long * outError)，inData是输入参数，指向输入数据的内存，ioLength在函数调用时赋值为输入数据的长度，函数返回时被赋值为实际完成转换的数据长度，outError为输出参数，用于指示

函数调用过程中发生的错误码。

笔者的编程风格大致就是这样，希望读者能够理解。重新回到DirectShow话题上来。开发DirectShow应用程序需要了解一些COM知识，主要是组件的“使用”问题。而开发DirectShow Filter则需要更多的COM知识，主要是组件的“实现”问题。在这里，笔者最后还要强调一下VC开发环境的配置。这一步是很重要的，因为如果开发环境没有配置好，很容易出现一些低级错误。（假设DirectX SDK安装在C:\DXSDK目录下。）

1. 编译基类源代码，至少生成两个静态库文件——打开C:\DXSDK\Samples\C++\DirectShow\BaseClasses\baseclasses.dsw，Debug版本生成strmbasd.lib，Release版本生成strmbase.lib。
2. 配置VC的编译环境：Include目录和Lib目录。执行VC的菜单命令Tools | Options，在随后弹出的对话框中单击Directories标签，在Show directories for下拉列表中选择Include files，然后配置如下：（注意，务必将DirectX SDK的目录放在标准的VC目录之前。）

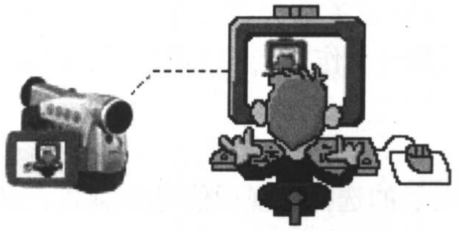
```
C:\DXSDK\Include
C:\DXSDK\SAMPLES\C++\DIRECTSHOW\BASECLASSES
C:\DXSDK\SAMPLES\C++\COMMON\INCLUDE
C:\Program Files\Microsoft Visual Studio\VC98\INCLUDE
C:\Program Files\Microsoft Visual Studio\VC98\MFC\INCLUDE
C:\Program Files\Microsoft Visual Studio\VC98\ATL\INCLUDE
```

再在Show directories for下拉列表中选择Library files，配置如下：

```
C:\DXSDK\Lib
C:\DXSDK\SAMPLES\C++\DIRECTSHOW\BASECLASSES\DEBUG
C:\DXSDK\SAMPLES\C++\DIRECTSHOW\BASECLASSES\RELEASE
C:\PROGRAM FILES\MICROSOFT SDK\LIB
C:\Program Files\Microsoft Visual Studio\VC98\LIB
C:\Program Files\Microsoft Visual Studio\VC98\MFC\LIB
```

3. 配置DirectShow应用程序开发项目需要连接的库文件。执行VC的菜单命令Project | Settings，在随后弹出的对话框中单击Link标签，在Object/library modules文本框中，Debug版本输入strmbasd.lib Winmm.lib，Release版本输入strmbase.lib Winmm.lib。
4. 如果安装的DirectX SDK的版本是9.0以前的，请确认在编译应用程序的Debug版本之前已经定义了DEBUG宏。执行VC的菜单命令Project | Settings，在随后弹出的对话框中单击C/C++标签，在Category下拉列表中选择Preprocessor，然后确认Preprocessor definitions文本框中有DEBUG（如果没有就加上）。

另外，应用程序在调用任何COM库函数之前，务必调用CoInitialize或CoInitializeEx进行COM库的初始化（一般只需在程序启动时调用一次）；在结束所有COM操作之后，调用CoUninitialize进行反初始化（一般在程序退出之前调用一次）。当程序中有多个线程都要使用COM库函数时，每个线程都要进行初始化和反初始化。总之，要保证CoUninitialize和CoInitialize（或CoInitializeEx）调用的一一配对。



2 音视频采集

2.1 功能介绍

音视频采集是DirectShow最基本的应用之一。据一份网上调查表明，DirectShow开发者中的60%都在做音视频采集相关的应用。所谓采集，通常是指将模拟信号采样生成数字信号，经过计算机处理后再现或存储到数字介质上。由于采集设备性能的差异以及兼容性问题，加上多媒体本身巨大的数据量，一般来说，采集的任务比较繁重，占用的系统资源也比较多，处理起来相当棘手！但自从有了DirectShow，情况就发生了改变——原来，采集也可以如此简单！

本章将要实现一个音视频采集的演示程序AVCap，它的主界面如图2.1所示。

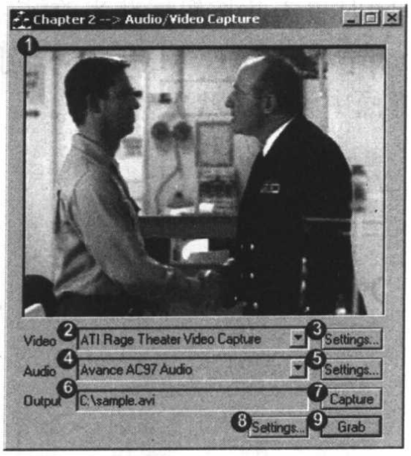


图 2.1　AVCap 程序主界面

① 视频显示窗口

显示采集设备输出的预览图像。如果系统中安装有视频采集设备，则该窗口中在程序启动之后就会有视频显示。

② 系统中所有的视频采集设备

系统中安装的所有符合DirectShow标准的视频采集设备都会在这个下拉列表框中显示。当存在多个采集设备时，可以选择其中任意一个作为当前工作的视频采集设备。

③ 视频参数设置

针对当前视频采集设备的参数设置，包括输入端子的选择、视频信号的制式、图像的显示参数（亮度、色调、饱和度、对比度等）、电视选台等。设置界面如图2.2所示。

④ 系统中所有的音频采集设备

系统中安装的所有符合DirectShow标准的音频采集设备都会在这个下拉列表框中显示。当存在多个采集设备时，可以选择其中任意一个作为当前工作的音频采集设备。

⑤ 音频参数设置

针对当前音频采集设备的参数设置，包括输入端子的选择、当前端子的合成音量大小等。设置界面如图2.5所示。

⑥ 采集输出的媒体文件

当进行采集到文件保存时，输出的文件名路径可以在这里设置。

⑦ 采集启动按钮

单击此按钮，将采集输出到⑥中指定的文件。此时，该按钮显示为Stop，再次单击后可停止采集。之后恢复为采集启动前的预览模式。

⑧ 采集输出参数设置

对采集到文件保存进行的一些参数设置，如输出图像的尺寸、视频的压缩方法、即时抓图文件的保存路径等。设置界面如图2.6所示。

⑨ 即时抓图按钮

进行视频预览时，可以随时单击此按钮捕获当前帧，保存为位图文件。位图文件名一般为hqHH-MM-SS.bmp，其中HH、MM、SS分别为当前时刻的时、分、秒；文件路径可以在图2.6的界面中设置（默认为C:\）。

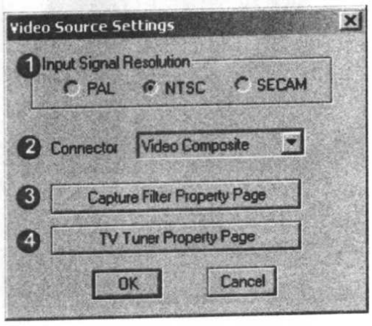


图 2.2　视频参数设置界面

① 输入视频信号的制式

选择一种电视制式（PAL、NTSC或SECAM）设置给采集设备，以便采集设备对输入信号进行正确的解码（否则图像显示不正常，对比效果见图2.14）。这部分的功能同样可以在③中完成。

② 视频输入端子

为采集设备选择一种视频输入端子，通常可以是AV复合端子、S-Video端子、电视接收等。具体选择哪种输入端子，取决于实际信号的输入（连接）方式。

③ 采集设备Filter属性页的显示按钮

采集设备Filter上暴露了一系列接口，用于各种参数（包括视频解码的制式、图像的亮度、对比度、色调、饱和度、锐利度等）的调整。这些接口在Filter的属性页中得到了应用。也就是说，用户可以通过属性页来改变上述这些参数。采集设备Filter的属性页如图2.3所示。

④ TV Tuner Filter属性页的显示按钮

有一些采集设备带有电视接收功能。电视接收由TV Tuner Filter来完成，在它的属性页上可以完成自动搜台、频道设定等功能。TV Tuner Filter的属性页如图2.4所示。

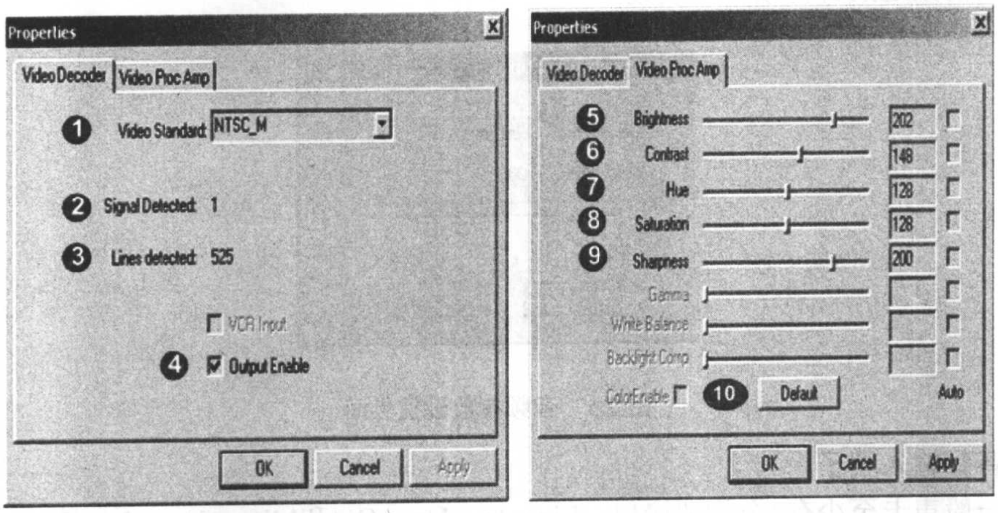


图 2.3　视频采集设备 Filter 的属性页

① 当前视频信号的制式　　② 指示是否探测到信号

③ 输入图像的水平扫描线数　　④ 是否允许采集卡输出

⑤ 采集图像的亮度调整　　⑥ 采集图像的对比度调整

⑦ 采集图像的色调调整　　⑧ 采集图像的饱和度调整

⑨ 采集图像的锐利度调整　　⑩ 图像参数恢复为默认值

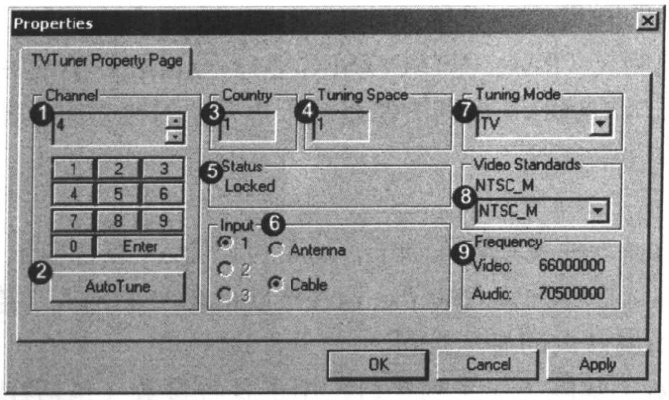


图 2.4 TV Tuner Filter 的属性页

① 当前观看的频道号

② 自动搜台按钮

③ 电视接收所在地的国家代码

④ 调谐空间（一般与国家代码对应）

⑤ 指示当前频道信号是否稳定

⑥ 输入端子（天线或电缆）

⑦ 调谐模式（电视或电台广播）

⑧ 接收电视的信号制式

⑨ 当前频道的音视频信号载频

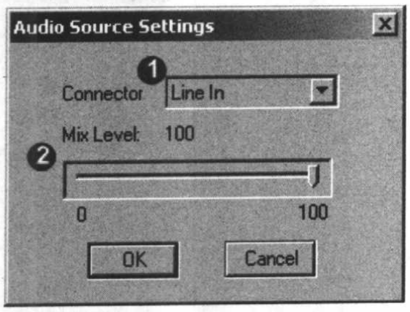


图 2.5 音频参数设置界面

① 音频输入端子

一般声卡至少有3个物理接口：Line In、Line Out和Microphone In。其中，Line In通常用于音频的采集，Line Out用于音频数据输出到音箱播放，Microphone In接受麦克风的输入。程序中选择好输入端子后，应确认对应的物理接口连接是否正常。

② 当前输入端子的合成音量大小

大部分声卡具有混音的功能。这里可以设置当前输入端子参与混音的音量大小。

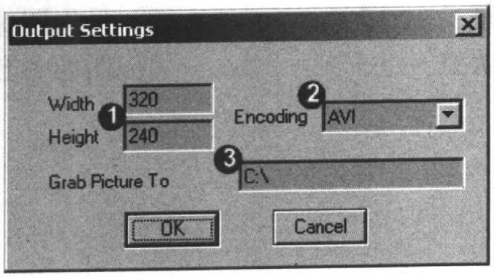


图 2.6 采集输出参数设置界面

① 采集时输出图像的宽度和高度
设置采集设备Filter输出图像的尺寸，默认是320 x 240，但可以调整到更大。考虑到采集的实时性要求，当②中视频编码采用DivX时，输出图像的尺寸限定为320 x 240（否则采集时占用过多系统资源，丢帧现象比较严重）。

② 视频编码格式
采集保存到AVI文件时，视频的编码格式可以在这里设置。如果保存为普通的AVI文件，视频图像的格式可能是RGB（如RGB32、RGB24、RGB565、RGB555等），也可能是YUV（如UYVY、YUY2等）。如果视频采用DivX压缩，请首先确认系统中已经安装了DivX的编码Filter。

③ 即时抓图文件的存储路径
通过AVCap主界面上的Grab按钮进行即时抓图时，生成的图片文件放到这个路径之下。

AVCap程序的界面大致就是这样，下面将逐一介绍这些功能的具体实现。

2.2 通用采集

2.2.1 采集设备枚举

AVCap程序要运行在不同的计算机、不同的操作系统上，但目标系统的硬件配置情况是不确定的，AVCap程序如何来使用这些不可预知的采集设备呢？答案是使用系统枚举。

在DirectShow框架下，参与工作的基本单元是Filter。采集设备也不例外。DirectShow使用特殊的包装Filter对它们进行了包装——WDM驱动模型的采集设备使用WDM Video Capture Filter（实现文件为kswdmcap.ax）、VFW驱动模型的采集设备使用VFW Capture Filter（实现文件为qcap.dll）。只要采集设备正确安装，DirectShow就能把它包装成一个Filter，并且在一定的类型目录下注册。换句话说，应用程序只要枚举特定的类型目录，就能知道系统中安装有多少个、以及何种类型的采集设备。

DirectShow定义了很多类型目录供不同的Filter注册。这里关心的音频采集设备注册在Audio Capture Sources目录下，视频采集设备注册在Video Capture Sources目录下。而在上述两个目录下注册的采集设备一般也会在WDM Streaming Capture Devices目录下注册。有时候，WDM Streaming Capture Devices还会包含一些既不在Audio Capture Sources下也不在Video Capture Sources下注册的采集设备。类型目录都用一个CLSID来表示，表2.1给出了音视频采集设备Filter注册的类型目录及其CLSID的对应关系。

表2.1 音视频采集设备Filter的注册目录

类型目录名	类型目录的CLSID	类型目录的MERIT
Audio Capture Sources	CLSID_AudioInputDeviceCategory	MERIT_DO_NOT_USE
Video Capture Sources	CLSID_VideoInputDeviceCategory	MERIT_DO_NOT_USE
WDM Streaming Capture Devices	AM_KSCATEGORY_CAPTURE	MERIT_DO_NOT_USE

DirectShow提供了一个专门的系统枚举组件（CLSID_SystemDeviceEnum）。枚举的大致过程如下：

（1）使用CoCreateInstance函数创建一个系统枚举组件对象，并获得ICreateDevEnum接口。

（2）使用接口方法ICreateDevEnum::CreateClassEnumerator为指定的类型目录创建一个枚举器，并获得IEnumMoniker接口。

（3）使用接口方法IEnumMoniker::Next枚举指定类型目录下所有的设备标识（Device Moniker）。每个设备标识对象上都实现了IMoniker接口。

（4）调用IMoniker::BindToStorage之后就可以访问设备标识的属性集，比如得到设备的显示名字（Display Name）、友好名字（Friendly Name）等。

（5）调用IMoniker::BindToObject可以将设备标识绑定成一个DirectShow Filter，随后调用IFilterGraph::AddFilter加入到Filter Graph中就可以参与工作了。

提示：DirectShow中的设备通常都有两个名字：显示名字和友好名字。前者一般是一个掺杂有很多信息的冗长字符串，如@device:cm:{33D9A762-90C8-11D0-BD43-00A0C911CE86}\Creative Sound Blaster PCI128，因此不适合直接显示给用户。而后者是对设备最直观的描述，如Creative Sound Blaster PCI128，适合在用户界面上显示。

在具体实现音视频采集设备的枚举时，AVCap程序定义了一个类CDSDevice来描述一个特定的设备。这个类具有3个属性：mType表示采集设备的类型（设备类型在GlobalDefs.h文件中定义），mFriendlyName表示采集设备的友好名字，mDisplayName表示采集设备的显示名字。CDSDevice类的完整实现如下：

```
//
// GlobalDefs.h
//
typedef enum
{
    DT_DV = 0,              // 数码摄像机
    DT_Analog_WDM,          // WDM视频采集设备
    DT_Analog_VFW,          // VFW视频采集设备
    DT_Analog_Audio,        // 音频采集设备
    DT_Unknown              // 未知设备
} Device_Type;

//
// CDSDevice.h
//

#ifndef __H_CDSDevice__
#define __H_CDSDevice__

#include <afxtempl.h>
#include "GlobalDefs.h"
```

```
class CDSDevice
{
private:
    Device_Type    mType;           // 采集设备的类型
    CString        mFriendlyName;   // 采集设备的友好名字
    CString        mDisplayName;    // 采集设备的显示名字
public:
    CDSDevice();
    ~CDSDevice();
    // 赋值操作符重载
    CDSDevice& operator = (const CDSDevice&);
    // 设置/获取采集设备的类型
    void SetDevideType(Device_Type inType);
    Device_Type GetDeviceType(void);
    // 判断当前设备是不是数码摄像机
    BOOL IsDVDevice(void);
    // 设置/获取采集设备的友好名字
    void SetDeviceFriendlyName(const char * inName);
    CString GetDeviceFriendlyName(void);
// 设置/获取采集设备的显示名字
    void SetDeviceDisplayName(const char * inName);
    CString GetDeviceDisplayName(void);
};

// 定义一个列表类型，用于保存系统中所有的音视频采集设备信息
typedef CList<CDSDevice, CDSDevice&> DEVICE_LIST;

#endif // __H_CDSDevice__

//
// CDSDevice.cpp
//

#include "stdafx.h"
#include "CDSDevice.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////////////////////////////////////////////////////
CDSDevice::CDSDevice()
{
    // 类成员的初始化
    mType         = DT_Unknown;
    mFriendlyName = "";
    mDisplayName  = "";
}
CDSDevice::~CDSDevice()
```

```
{
}
// 赋值操作符的重载
CDSDevice& CDSDevice::operator = (const CDSDevice& inDevice)
{
    mType            = inDevice.mType;
    mFriendlyName    = inDevice.mFriendlyName;
    mDisplayName     = inDevice.mDisplayName;
    return *this;
}
// 设置采集设备的类型
void CDSDevice::SetDevideType(Device_Type inType)
{
    mType = inType;
}
// 获取采集设备的类型
Device_Type CDSDevice::GetDeviceType(void)
{
    return mType;
}
// 判断当前设备是不是数码摄像机
BOOL CDSDevice::IsDVDevice(void)
{
    return (mType == DT_DV);
}
// 设置采集设备的友好名字
void CDSDevice::SetDeviceFriendlyName(const char * inName)
{
    mFriendlyName = inName;
}
// 获取采集设备的友好名字
CString CDSDevice::GetDeviceFriendlyName(void)
{
    return mFriendlyName;
}
// 设置采集设备的显示名字
void CDSDevice::SetDeviceDisplayName(const char * inName)
{
    mDisplayName = inName;
}
// 获取采集设备的显示名字
CString CDSDevice::GetDeviceDisplayName(void)
{
    return mDisplayName;
}
```

因为系统中可能安装了多个采集设备，所以最好使用一个列表来保存这些采集设备的信息。AVCap程序定义了两个类：CAudioDevices和CVideoDevices，分别用于维护音频采集设备和视频采集设备的列表。这两个类的实现很类似，下面仅以CAudioDevices为例加以说明。

```
//
// CAudioDevices.h
//

#ifndef __H_CAudioDevices__
#define __H_CAudioDevices__

#include "CDSDevice.h"

class CAudioDevices
{
public:
    DEVICE_LIST        mList;                   // 设备列表
public:
    static CAudioDevices * Instance(void);      // 静态成员函数
    static void Release(void);
protected:
    CAudioDevices();
    ~CAudioDevices();
    // 枚举得到系统中所有的音频采集设备
    void Populate(void);
private:
    static CAudioDevices * sDevices;            // 私有静态成员
};

#endif // __H_CAudioDevices__

//
// CAudioDevices.cpp
//

#include "stdafx.h"
#include <streams.h>
#include "CAudioDevices.h"
#include "UDsUtils.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

CAudioDevices * CAudioDevices::sDevices = NULL;
//////////////////////////////////////////////////////////////////////////
CAudioDevices::CAudioDevices()
{
    // 在构造函数中完成系统枚举
    Populate();
}
CAudioDevices::~CAudioDevices()
{
    // 清空设备列表
```

```
        mList.RemoveAll();
    }
    // CAudioDevices类在系统中只允许存在一个实例，这个惟一实例通过静态成员函数Instance
    // 来访问
    CAudioDevices * CAudioDevices::Instance(void)
    {
        if (sDevices == NULL)
        {
            sDevices = new CAudioDevices();
        }
        return sDevices;
    }
    // 在系统退出之前，应调用下面的静态成员函数来释放CAudioDevices类的惟一实例
    void CAudioDevices::Release(void)
    {
        if (sDevices)
        {
            delete sDevices;
            sDevices = NULL;
        }
    }
    // 枚举得到系统中所有的音频采集设备
    void CAudioDevices::Populate(void)
    {
        mList.RemoveAll();
        UDsUtils::QueryAudioCaptureDevices(mList);
    }
```

小知识：单件模式

CAudioDevices和CVideoDevices两个类的设计都用到了单件模式。那么，何为单件模式？为什么这两个类要使用单件模式呢？

所谓单件（Singleton），就是说某个类在系统中只存在一个实例。当某个类只能有一个实例，而且要求客户可以从一个众所周知的访问点访问它时，就应该考虑采用单件模式。通常情况下，单件类自身负责保存它的惟一实例，并且使用一个静态成员函数提供全局的访问点。（一个全局变量同样使得一个对象随时随地可以被访问，但它不能防止客户实例化多个对象，另外还污染了名字空间。）

系统的硬件配置信息通常只需在程序启动的时候进行一次性的搜集。因此，用于维护系统中音视频采集设备信息的CAudioDevices和CVideoDevices两个类都采用了单件模式。

在C++中，可以为单件类（类名假设为Singleton）定义一个静态成员函数Instance，以提供这个类操作的全局访问点。Singleton类还定义了一个静态成员变量_instance，用于保存一个指向它的惟一实例的指针。

Singleton类的定义如下：

```
class Singleton
{
public:
    static Singleton * Instance();
protected:
```

```
    Singleton();
private:
    static Singleton * _instance;
};
```

相应的实现如下:

```
Singleton * Singleton::_instance = 0;
Singleton * Singleton::Instance()
{
    if (_instance == 0)
    {
        _instance = new Singleton();
    }
    return _instance;
}
```

客户仅通过Singleton::Instance函数访问这个单件。变量_instance初始化为0，而静态成员函数Instance返回该变量值，如果其值为0，则用惟一实例初始化它。Instance使用惰性（Lazy）初始化；它的返回值直到第一次访问时才被创建和保存。另外，注意Singleton类的构造函数是保护型的（protected），试图直接实例化Singleton类的客户将得到一个编译时的错误信息。这就保证了仅有一个实例可以被创建。

更多关于单件模式的介绍，请读者参考《设计模式——可复用面向对象软件的基础》一书的3.5节。

CAudioDevices类在进行系统枚举（参见Populate函数）的时候，实际上调用了UDsUtils类的一个静态成员函数QueryAudioCaptureDevices，而这个函数才是系统枚举功能真正实现的地方。（UDsUtils一个工具类，实现的都是一些相对独立的功能。）

```
//
// UDsUtils.cpp
//
// 欲得到系统中所有的视频采集设备，则枚举CLSID_VideoInputDeviceCategory目录
BOOL UDsUtils::QueryVideoCaptureDevices(DEVICE_LIST & outDevices)
{
    return QueryDeviceCategory(CLSID_VideoInputDeviceCategory,
            outDevices);
}
// 欲得到系统中所有的音频采集设备，则枚举CLSID_AudioInputDeviceCategory目录
BOOL UDsUtils::QueryAudioCaptureDevices(DEVICE_LIST & outDevices)
{
    return QueryDeviceCategory(CLSID_AudioInputDeviceCategory,
            outDevices);
}
// 给定一个类型目录，枚举该目录下所有的设备，并放到指定列表中保存
BOOL UDsUtils::QueryDeviceCategory(GUID inCategory, DEVICE_LIST &
        outDevices)
{
    HRESULT hr = NOERROR;
    ICreateDevEnum * enumHardware = NULL;
    // 创建一个系统枚举组件对象
```

```
hr = CoCreateInstance(CLSID_SystemDeviceEnum, NULL, CLSCTX_ALL,
    IID_ICreateDevEnum, (void**)&enumHardware);
if (FAILED(hr))
{
    return FALSE;
}

// 为指定的类型目录创建一个枚举器
IEnumMoniker * enumMoniker = NULL;
hr = enumHardware->CreateClassEnumerator(inCategory, &enumMoniker, 0);
if (enumMoniker)
{
    enumMoniker->Reset();

    char displayName[1024];
    char friendlyName[256];
    ULONG fetched = 0;
    IMoniker * moniker = NULL;
    // 逐一查询类型目录下所有的设备
    while(SUCCEEDED(enumMoniker->Next(1, &moniker, &fetched)) &&
        fetched)
    {
        if (moniker)
        {
            WCHAR * wzDisplayName     = NULL;
            IPropertyBag * propertyBag = NULL;
            IBaseFilter * filter      = NULL;
            VARIANT       name;

            // 获取当前设备的显示名字
            hr = moniker->GetDisplayName(NULL, NULL, &wzDisplayName);
            if (SUCCEEDED(hr))
            {
                WideCharToMultiByte(CP_ACP, 0, wzDisplayName, -1,
                    displayName, 1024, "", NULL);
                CoTaskMemFree(wzDisplayName);
                hr = moniker->BindToStorage(0, 0, IID_IPropertyBag,
                    (void **)&propertyBag);
            }
            // 获取当前设备的友好名字
            if (SUCCEEDED(hr))
            {
                name.vt = VT_BSTR;
                hr = propertyBag->Read(L"FriendlyName", &name, NULL);
            }
            if (SUCCEEDED(hr))
            {
                friendlyName[0] = 0;
                WideCharToMultiByte(CP_ACP, 0, name.bstrVal, -1,
                    friendlyName, 256, NULL, NULL);
                // 将设备标识绑定为Filter形式
                hr = moniker->BindToObject(0, 0, IID_IBaseFilter,
```

```
                    (void**)&filter);
            }
            if (SUCCEEDED(hr))
            {
                // 保存当前设备信息
                CDSDevice  device;
                device.SetDeviceDisplayName(displayName);
                device.SetDeviceFriendlyName(friendlyName);
                // 识别当前设备的类型
                device.SetDevideType(DecideDeviceType(filter));
                // 将当前设备加入到列表中
                outDevices.AddTail(device);
            }

            // 释放使用过的接口
            if (filter)
            {
                filter->Release();
                filter = NULL;
            }
            if (propertyBag)
            {
                propertyBag->Release();
                propertyBag = NULL;
            }
            moniker->Release();
        }
    }
    enumMoniker->Release();
  }
  enumHardware->Release();

  return TRUE;
}
// 识别一个设备的类型：数码摄像机、视频采集设备、音频采集设备等
// 通过查询给定Filter上的一些特殊接口来实现
Device_Type UDsUtils::DecideDeviceType(IBaseFilter * inFilter)
{
    // 如果该Filter上支持IAMExtTransport接口，则认为它是数码摄像机
    IAMExtTransport * pAMExtTransPost = NULL;
    inFilter->QueryInterface(IID_IAMExtTransport, (void
            **)&pAMExtTransPost);
    if (pAMExtTransPost)
    {
        pAMExtTransPost->Release();
        return DT_DV;
    }
    // 如果该Filter上支持IAMAnalogVideoDecoder接口，则认为它是WDM视频采集设备
    IAMAnalogVideoDecoder * pDecoder = NULL;
    inFilter->QueryInterface(IID_IAMAnalogVideoDecoder, (void
            **)&pDecoder);
    if (pDecoder)
```

```
    {
        pDecoder->Release();
        return DT_Analog_WDM;
    }
    // 如果该Filter上支持IAMVfwCaptureDialogs接口，则认为它是VFW视频采集设备
    IAMVfwCaptureDialogs * pVfwDlgs = NULL;
    inFilter->QueryInterface(IID_IAMVfwCaptureDialogs, (void
            **)&pVfwDlgs);
    if (pVfwDlgs)
    {
        pVfwDlgs->Release();
        return DT_Analog_VFW;
    }
    // 如果该Filter上支持IAMAudioInputMixer接口，则认为它是音频采集设备
    IAMAudioInputMixer * pAudioMixer = NULL;
    inFilter->QueryInterface(IID_IAMAudioInputMixer, (void
            **)&pAudioMixer);
    if (pAudioMixer)
    {
        pAudioMixer->Release();
        return DT_Analog_Audio;
    }
    // 这是一个无法识别的设备
    return DT_Unknown;
}
```

既然CAudioDevices和CVideoDevices通过系统枚举搜集到了系统中的所有音视频采集设备信息，则可以将这些信息在AVCap程序的主界面上显示出来（使用两个组合框）。这部分工作在AVCap程序的主对话框类CAVCapDlg中完成。

```
//
// AVCapDlg.h
//

#include "CLiveCapture.h"

class CAVCapDlg : public CDialog
{
// Construction
public:
    CAVCapDlg(CWnd* pParent = NULL);    // standard constructor

// Dialog Data
    //{{AFX_DATA(CAVCapDlg)
    enum { IDD = IDD_AVCAP_DIALOG };
    CComboBox   mComboAudioDevices;
    CComboBox   mComboVideoDevices;
    //}}AFX_DATA

    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CAVCapDlg)
    protected:
```

```
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:
    HICON m_hIcon;
    CLiveCapture  mLiveCapture;

    void FillAudioDevices(void);
    void FillVideoDevices(void);

    // Generated message map functions
    //{{AFX_MSG(CAVCapDlg)
    afx_msg void OnDestroy();
    virtual BOOL OnInitDialog();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

//
// AVCapDlg.cpp
//

#include "CVideoDevices.h"
#include "CAudioDevices.h"

BEGIN_MESSAGE_MAP(CAVCapDlg, CDialog)
    //{{AFX_MSG_MAP(CAVCapDlg)
    ON_WM_DESTROY()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

// 主对话框的初始化函数
BOOL CAVCapDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // Set the icon for this dialog.  The framework does this automatically
    //  when the application's main window is not a dialog
    SetIcon(m_hIcon, TRUE);         // Set big icon
    SetIcon(m_hIcon, FALSE);        // Set small icon

    // TODO: Add extra initialization here
    FillAudioDevices(); // 填写音频设备信息
    FillVideoDevices(); // 填写视频设备信息

    // 其他初始化代码（省略）
    // ……

    return TRUE;  // return TRUE  unless you set the focus to a control
}
// 主对话框销毁之前释放资源
```

```
void CAVCapDlg::OnDestroy()
{
    // 其他反初始化代码（省略）
    // ……

    // 销毁保存音视频采集设备信息的单件对象
    CVideoDevices::Release();
    CAudioDevices::Release();

    CDialog::OnDestroy();
}
// 将枚举得到的所有音频设备填入主对话框上的音频设备组合框
void CAVCapDlg::FillAudioDevices(void)
{
    // 清空组合框中的内容
    mComboAudioDevices.ResetContent();
    // 访问音频采集设备列表
    POSITION pos = CAudioDevices::Instance()->mList.GetHeadPosition();
    while (pos)
    {
        // 将所有音频采集设备的名字加入组合框
        CDSDevice device = CAudioDevices::Instance()->mList.GetNext(pos);
        mComboAudioDevices.AddString(device.GetDeviceFriendlyName());
    }
    // 自动选择第一个为当前工作的音频采集设备
    if (mComboAudioDevices.GetCount() > 0)
    {
        mComboAudioDevices.SetCurSel(0);
        CString deviceName;
        mComboAudioDevices.GetLBText(0, deviceName);
        mLiveCapture.SetAudioDevice(deviceName);
    }
}
// 将枚举得到的所有视频设备填入主对话框上的视频设备组合框
void CAVCapDlg::FillVideoDevices(void)
{
    // 清空组合框中的内容
    mComboVideoDevices.ResetContent();
    // 访问视频采集设备列表
    POSITION pos = CVideoDevices::Instance()->mList.GetHeadPosition();
    while (pos)
    {
        // 将所有视频采集设备的名字加入组合框
        CDSDevice device = CVideoDevices::Instance()->mList.GetNext(pos);
        mComboVideoDevices.AddString(device.GetDeviceFriendlyName());
    }
    // 自动选择第一个为当前工作的视频采集设备
    if (mComboVideoDevices.GetCount() > 0)
    {
        mComboVideoDevices.SetCurSel(0);
        CString deviceName;
        mComboVideoDevices.GetLBText(0, deviceName);
```

```
            mLiveCapture.SetVideoDevice(deviceName);
        }
    }
```

2.2.2 采集设备创建

采集设备必须以Filter的形式才能参与到DirectShow中，进而与其他Filter协同工作。采集设备Filter的创建实质上也是一个系统枚举过程。当用户选定了一个音频或视频采集设备后，可以得到这个设备的友好名字或者显示名字，然后就可以以这个名字为参数去创建它。具体实现时，就是枚举特定目录下的所有设备，取出它们的名字与参数进行比较，如果匹配则将当前设备标识绑定为一个Filter的形式。

采集设备的创建仍然在工具类UDsUtils中实现，参考如下：

```
//
// UDsUtils.cpp
//
// 指定视频采集设备的友好名字，为它创建一个Filter
IBaseFilter * UDsUtils::CreateVideoDevice(const char * inFriendlyName)
{
    return CreateHardwareFilter(CLSID_VideoInputDeviceCategory,
            inFriendlyName);
}
// 指定音频采集设备的友好名字，为它创建一个Filter
IBaseFilter * UDsUtils::CreateAudioDevice(const char * inFriendlyName)
{
    return CreateHardwareFilter(CLSID_AudioInputDeviceCategory,
            inFriendlyName);
}
// 根据设备的友好名字，创建一个代表该设备的Filter
IBaseFilter * UDsUtils::CreateHardwareFilter(GUID inCategory, const char *
        inFriendlyName)
{
    // 创建一个系统枚举组件对象
    ICreateDevEnum * enumHardware = NULL;
    HRESULT hr = CoCreateInstance(CLSID_SystemDeviceEnum, NULL, CLSCTX_ALL,
        IID_ICreateDevEnum, (void**)&enumHardware);
    if (FAILED(hr))
    {
        return NULL;
    }

    IBaseFilter * hardwareFilter = NULL;
    IEnumMoniker * enumMoniker   = NULL;
    // 为指定的目录创建枚举器
    hr = enumHardware->CreateClassEnumerator(inCategory, &enumMoniker, 0);
    if (enumMoniker)
    {
        enumMoniker->Reset();

        ULONG      fetched = 0;
```

```
        IMoniker * moniker = NULL;
        char friendlyName[256];

        // 枚举得到该目录下所有的设备，逐个进行名字匹配
        while (!hardwareFilter && SUCCEEDED(enumMoniker->Next(1, &moniker,
               &fetched)) && fetched)
        {
            if (moniker)
            {
                IPropertyBag * propertyBag = NULL;
                VARIANT       name;
                friendlyName[0] = 0;
                hr = moniker->BindToStorage(0, 0, IID_IPropertyBag, (void
                        **)&propertyBag);
                // 读取设备的友好名字
                if (SUCCEEDED(hr))
                {
                    name.vt = VT_BSTR;
                    hr = propertyBag->Read(L"FriendlyName", &name, NULL);
                }
                if (SUCCEEDED(hr))
                {
                    WideCharToMultiByte(CP_ACP, 0, name.bstrVal, -1,
                        friendlyName, 256, NULL, NULL);
                    // 如果当前设备的友好名字与用户指定的设备名字相同，
                    // 则将当前设备标识绑定为Filter形式
                    if (strcmp(friendlyName, inFriendlyName) == 0)
                    {
                        moniker->BindToObject(0, 0, IID_IBaseFilter,
                            (void**)&hardwareFilter);
                    }
                }

                // 释放使用过的接口
                if (propertyBag)
                {
                    propertyBag->Release();
                    propertyBag = NULL;
                }
                moniker->Release();
            }
        }
        enumMoniker->Release();
    }
    enumHardware->Release();

    return hardwareFilter;
}
```

提示：AVCap程序分别为音频采集设备和视频采集设备提供了包装类：CAudioCaptureFilter和CVideoCaptureFilter。采集设备的创建实际上是在这两

个类的CreateFilter函数中完成的——分别调用UDsUtils::CreateAudioDevice函数和UDsUtils::CreateVideoDevice函数。

CreateHardwareFilter函数根据设备的友好名字去创建Filter。当然，通过设备的显示名字也可以去创建Filter。UDsUtils 类中提供了另外一个函数CreateHardwareFilter2 来实现这个功能。这个函数成功创建了Filter之后，还能返回这个设备的友好名字（这个功能将会在2.2.8节中用到）。

```
//
// UDsUtils.cpp
//
// 通过设备的显示名字来创建设备，并且返回这个设备的友好名字
IbaseFilter * UDsUtils::CreateDevice(BOOL inIsVideo, const char *
      inDisplayName, char * outFriendlyName)
{
    GUID guid = inIsVideo ? CLSID_VideoInputDeviceCategory :
        CLSID_AudioInputDeviceCategory;
    return CreateHardwareFilter2(guid, inDisplayName, outFriendlyName);
}
// 根据设备的显示名字创建一个代表该设备的Filter，并且返回该设备的友好名字
IBaseFilter * UDsUtils::CreateHardwareFilter2(GUID inCategory, const char
        * inDisplayName,  char * outFriendlyName)
{
    // 创建一个系统枚举组件对象
    ICreateDevEnum * enumHardware = NULL;
    HRESULT hr = CoCreateInstance(CLSID_SystemDeviceEnum, NULL, CLSCTX_ALL,
        IID_ICreateDevEnum, (void**)&enumHardware);
    if (FAILED(hr))
    {
        return NULL;
    }

    IBaseFilter * hardwareFilter = NULL;
    IEnumMoniker * enumMoniker   = NULL;
    // 为指定的目录创建枚举器
    hr = enumHardware->CreateClassEnumerator(inCategory, &enumMoniker, 0);
    if (enumMoniker)
    {
        enumMoniker->Reset();

        ULONG     fetched = 0;
        IMoniker * moniker = NULL;
        char displayName[1024];

        // 枚举得到目录下所有的设备，逐个进行名字匹配
        while (!hardwareFilter && SUCCEEDED(enumMoniker->Next(1, &moniker,
                &fetched)) && fetched)
        {
            if (moniker)
            {
                WCHAR * wzDisplayName = NULL;
```

```
                // 获得设备的显示名字
                hr = moniker->GetDisplayName(NULL, NULL, &wzDisplayName);
                if (SUCCEEDED(hr))
                {
                    WideCharToMultiByte(CP_ACP, 0, wzDisplayName, -1,
                        displayName, 1024, "", NULL);
                    CoTaskMemFree(wzDisplayName);
                    // 判断设备的显示名字是否匹配
                    if (IsSameDevice(displayName, inDisplayName))
                    {
                        moniker->BindToObject(0, 0, IID_IBaseFilter,
                            (void**)&hardwareFilter);
                    }
                }
                // 如果成功创建了Filter，则获取该设备的友好名字
                if (outFriendlyName && hardwareFilter)
                {
                    IPropertyBag * propertyBag = NULL;
                    outFriendlyName[0] = 0;
                    if (SUCCEEDED(moniker->BindToStorage(0, 0,
                        IID_IPropertyBag, (void **)&propertyBag)))
                    {
                        VARIANT name;
                        name.vt = VT_BSTR;
                        if (SUCCEEDED(propertyBag->Read(L"FriendlyName",
                            &name, NULL)))
                        {
                            WideCharToMultiByte(CP_ACP, 0, name.bstrVal,
                                -1, outFriendlyName, 1024, NULL, NULL);
                        }
                        propertyBag->Release();
                    }
                }
                moniker->Release();
            }
        }
        enumMoniker->Release();
    }
    enumHardware->Release();

    return hardwareFilter;
}
// 判断两个显示名字指的是不是同一个设备
// 去掉显示名字"\\"之前的字符，然后忽略字母大小写进行比较
BOOL UDsUtils::IsSameDevice(const char * inDeviceName1, const char *
     inDeviceName2)
{
    CString temp1, temp2, deviceName1, deviceName2;

    temp1 = inDeviceName1;
    temp2 = inDeviceName2;
    // 在显示名字的字符串中找到"\\"的位置
```

```
    int backSlash1 = temp1.Find("\\\\", 0);
    int backSlash2 = temp2.Find("\\\\", 0);
    if (backSlash1 >= 0 && backSlash2 >= 0)
    {
        deviceName1 = temp1.Right(temp1.GetLength() - backSlash1);
        deviceName2 = temp2.Right(temp2.GetLength() - backSlash2);
        // 忽略字母大小写进行比较
        return (deviceName1.CompareNoCase(deviceName2) == 0);
    }
    else
    {
        return (temp1.CompareNoCase(temp2) == 0);
    }
    return FALSE;
}
```

提示：在热插拔设备接入或脱离计算机时，系统会发送一个WM_DEVICECHANGE消息给应用程序（的某个注册窗口），消息的参数中指出了这个设备的显示名字。但是，Windows给出的设备显示名字与DirectShow进行系统枚举获得的该设备的显示名字有点不一样。如前者可能是\\?\1394#A02D&10001#C00A10021888000#{65e8773d-8f56-11d0-a3b9-00a0c9223196}\GLOBAL，而后者可能是@device:pnp:\\?\1394#a02d&10001#c00a10021888000#{65e8773d8f-5611d0-a3b9-00a0c9223196}\global。因此，需要通过UDsUtils::IsSameDevice函数来判断两个显示名字指的是不是同一个设备：去掉两个显示名字的前缀，忽略字母大小写。

2.2.3 开始视频预览

采集设备的成功创建是万里长征迈出的第一步。如果想要进一步预览采集设备输出的视频图像，则必须为它创建一个完整的Filter Graph。采集设备的种类有很多，按照驱动模型可以分为WDM设备和VFW设备，而按照接口方式可以分为PCI设备、AGP设备、USB设备、1394设备等。不同的采集设备构建的Filter Graph也有所不同。如何使用一种统一的方法，为不同的采集设备创建不同的Filter Graph（也就是应用程序对各种采集设备的兼容性问题），是这一步最大的挑战。

典型的采集设备有一个Preview输出Pin和一个Capture输出Pin，前者主要用于视频图像的预览，而后者主要用于视频数据的采集。这种采集设备多数以PCI（或USB）接口接入计算机。图2.7为这种采集设备构建单纯预览用的Filter Graph。（有些采集设备没有Preview输出Pin，而只有一个Capture输出Pin，则预览时也可以将Capture输出Pin接到Video Renderer上。）

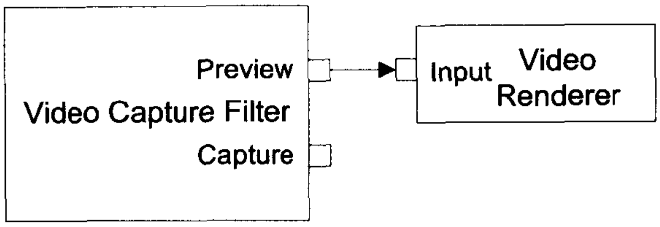


图 2.7 PCI 采集设备预览用的 Filter Graph

另一类采集设备使用了AGP接口。这种采集设备将采集功能与显卡的功能集成在一起，从而能极大地提高视频显示的效率。一般这种采集设备没有称为Preview的输出Pin，取而代之的是一个Video Port Pin（简称VP Pin）。图2.8为AGP采集设备构建单纯预览用的Filter Graph。值得注意的是，预览时并没有真正的视频数据从VP Pin输出（视频数据直接由采集模块送到显存）。此时，Overlay Mixer与Video Renderer之间当然也没有真正的视频数据流动，Video Renderer仅仅是作为一个视频窗口管理器而存在。

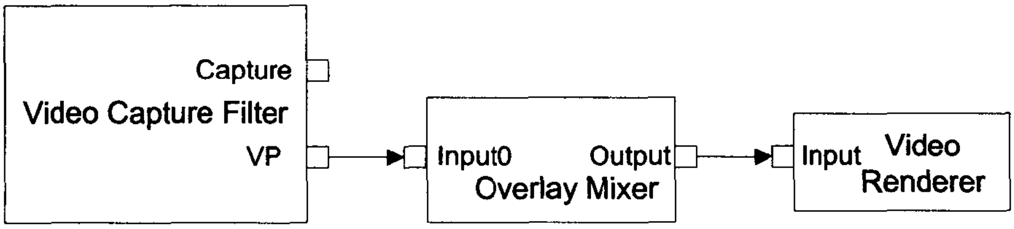


图 2.8 AGP 采集设备预览用的 Filter Graph

图2.8是带有VP Pin的采集设备预览用的、各个Windows操作系统上通用的Filter Graph。如果是在Windows XP下，另外还有一种连接方法可供选择，即使用Video Port Manager和VMR-7配对，如图2.9所示。值得注意的是，DirectX 9.0发布的VMR-9不支持与VP Pin的连接。

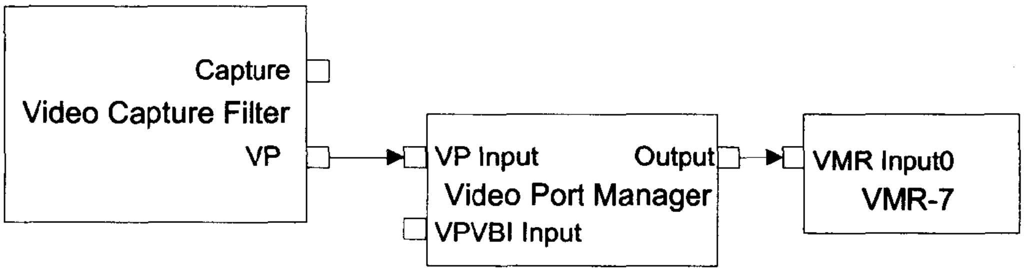


图 2.9 AGP 采集设备在 Windows XP 下可选的 Filter Graph

小知识：PC 总线简史

经常听人说他的计算机硬盘是IDE总线的，光驱是SCSI总线的，主板是PCI总线的，显卡是AGP总线的，不知道这些总线的名称各代表什么意思？是按什么来区分的？这么多的总线类型是否可以在同一台计算机上使用？

计算机的总线是计算机传输指令、数据和地址的线路，是计算机各部件联系的桥梁。一般来说，按照总线在计算机中的位置可以分为机器内部总线和机器外设总线两类。

机器内部总线是计算机内部各部件通信的总线，按照发展的过程分为以下几种：ISA总线、EISA总线、VESA总线、PCI总线和AGP总线等。ISA是用于286计算机的总线标准，EISA是用于386计算机的总线标准，VESA是用于486计算机的总线标准，PCI是用于586计算机及更高机型的总线标准，前三种总线已经被淘汰。AGP总线只负责控制芯片和AGP显卡之间的交互，可以和PCI总线共存。

PCI总线标准是1992年推出的。从原先8位的PC/XT、16位的ISA总线、32位的EISA、MCA、VL总线，到PCI总线，再到如今64位的PCI-64/66MHz、PCI-X 1.0和PCI-X 2.0，PC总线走过了一段漫长的发展历史。在这段历史中，PCI总线无疑是一颗最为耀眼的明星，就算在今天也还闪烁着璀璨的光芒。然而，PCI是开放型通用的内部系统总线，它不仅连接PCI显卡，还连接其他众多的PCI设备（如声卡、MPEG解压卡、网卡、电视卡等）。也就是说，PCI总线133MB/s的带宽不是给PCI显卡专用的，而要同时分给工作的所有PCI设备。这样在处理3D图形的过程中，当显卡要求通过PCI总线和系统进行数据交换时，其中的纹理数据需要占用很大的带宽，于是PCI总线就成了制约图形子系统和整个系统性能的瓶颈。在这种情况下，AGP总线便应运而生了。

AGP（Accelerated Graphics Port，图形加速端口）是在PCI图形接口的基础上发展而来的。然而它又是一种与PCI总线截然不同的图形接口。AGP完全独立于PCI总线之外，直接把显卡与主板控制芯片连在一起，使得3D图形数据避开了通过PCI总线传输的过程，从而很好地解决了低带宽PCI接口造成的系统瓶颈问题。1996年7月，AGP 1.0规范问世，可以有AGP 1X和AGP 2X两种模式，工作频率为66MHz，是PCI的2倍，而数据传输带宽分别达到了266MB/s和533MB/s，分别是PCI的2倍和4倍。尽管如此，由于显示芯片的发展速度实在是太快了，图形卡单位时间内所能处理的数据呈几何级数增长，AGP 1.0图形标准也难以满足技术进步所带来的更高的需求。于是Intel在1998年5月又发布了AGP 2.0规范，工作频率依然是66MHz，也就是目前主流的AGP 4X模式，它的数据传输带宽达到了1.066GB/s，数据传输能力大大地增强了。2002年9月，Intel再次发布了AGP 3.0规范，AGP 8X作为新一代AGP接口总线，其数据宽度仍然是32位，但工作频率达到了533MHz，使其传输带宽高达2.1GB/s，是原来AGP 4X的2倍。AGP 3.0的推出在一定程度上适应了当今CPU和GPU（图形工作站）的飞速发展，使图形子系统的性能和系统性能得到了提高。表2.2给出了目前几种PCI、AGP标准主要参数的比较。

表2.2 几种PCI、AGP标准主要参数的比较

性能参数	PCI 2.2	AGP 1X	AGP 2X	AGP 4X	AGP 8X
数据宽度	32位	32位	32位	32位	32位
工作频率	33MHz	66MHz	66MHz	66MHz	533MHz
传输速率	133MB/s	266MB/s	533MB/s	1.06GB/s	2.1GB/s

值得注意的是，在数据量（或者说带宽）的表示上，存在着位（bit）和字节（byte）两种表示方式，而且在数据的传输上也存在着按位传输和按字节传输两种不同的方式。众所周知，8位构成1字节，所以某些时候如果没有注意到上述的区别，就容易产生歧义。通常，我们按设备和接口的工作方式，分别用字节/秒（B/S）和位每秒（bps）来表示其传输

速率，如串口和USB接口都是按位来传输数据的，它们的传输速率就以bps来表示（USB 1.1的传输速率为12Mbps，USB 2.0的传输速率为480Mbps）；而并口和硬盘接口等是按字节来传输数据的，所以一般用B/S来表示其传输速率。遗憾的是，不少专业报刊都混淆了这两种表示方式。一个典型的例子就是关于IEEE 1394的传输速率，很多时候都被表示为400MB/s，很容易让人认为是每秒钟传输400兆字节，但只要仔细想一下就知道这在现在是不可能的，因为现在IEEE 1394设备还只是33MHz 、32位的PCI设备，而33MHz 、32位PCI总线的理论带宽为133MB/s，任何33MHz 、32位的PCI设备都不可能超越这个极限。实际上，IEEE 1394是一种串行接口，上面所标示的速率应为400兆位每秒，换算到字节表示就是50兆字节每秒。IEEE 1394传输速率正确的表示方式应该是400Mbps。

AGP总线作为一种局部总线，它的推出只是在一定程度上缓解了高速图形数据交换对整个系统总线的压力，这是一种治标不治本的做法。随着计算机技术（特别是CPU、存储器、显示设备）日新月异的高速发展，旧有的系统总线架构已经无法满足技术发展的需要。随着Intel全新一代的PCI Express规范（也称为3GIO，即第三代I/O技术）的推出，AGP 3.0可能是我们见到的最后的AGP规范了，AGP 8X也将是AGP最后的辉煌。

接下来再来简单介绍一下机器外设总线。机器外设总线是计算机内部与外设进行通信的总线，分为IDE总线、SCSI总线和USB总线等。IDE总线是PC机上用得最多的总线，其造价比较便宜。SCSI总线的速度比IDE总线要快得多，不过造价比较昂贵。IDE总线和SCSI总线一般只是用于硬盘、光驱和扫描仪等，而USB总线则可以用于更多的外设，且速度更快。一般来说，这三种外设总线是不可以混合使用的，除非安装了总线转换器。

构建牵涉到采集设备的Filter Graph时是有一些麻烦的。因此，DirectShow提供了一个专门的Capture Graph Builder组件（CLSID_CaptureGraphBuilder2）来辅助这种Filter Graph的构建。使用方法也很简单，只需创建这个组件并获得ICaptureGraphBuilder2接口，然后调用接口方法SetFiltergraph将IGraphBuilder （Filter Graph Manager上的接口）设置给它，接着，就可以调用ICaptureGraphBuilder2的其他接口方法继续Filter Graph的构建了。

虽然使用Capture Graph Builder组件能够很容易为采集设备构建完整的预览用的Filter Graph，但在AVCap程序中，考虑到以后的采集应用，当这个辅助组件完成了预览用的Filter Graph构建后，还是将采集设备Filter后面接着的所有Filter断开、删除。因为预览和采集两种应用，采集设备前端Filter链路的构建是相同的，而仅在后端有些差异：预览时只接Video Renderer（和Audio Renderer），而在采集时可能还要接音视频编码Filter，最终使用File Writer将数据写到文件中去。于是，AVCap程序将Filter链路抽象成两部分：采集设备Filter以前的（包括采集设备Filter本身）定义为输入部分，而采集设备Filter以后的定义为输出部分。两部分的Filter分开创建，然后再连接。在程序开发时，对于预览和采集两种应用，输入部分的构建是相同的，于是开发的重点就可以放在根据不同的应用定制不同的输出部分上。

读者可能有疑问，既然使用Capture Graph Builder组件构建的Filter链路最终还是被删除了，那为什么还要用这个组件呢？那是因为通常采集设备Filter之前还有其他的Filter链路需要构建，比如像图2.10中所示的：需要连接Crossbar Filter用于输入端子的选择，有些电视卡还要连接TV Tuner Filter等。这些Filter都是跟硬件相关联的，创建并连接它们是一个繁琐的过程。但是，调用ICaptureGraphBuilder2接口的RenderStream函数却可以轻松做到这一点，

何乐而不为呢？另外，ICaptureGraphBuilder2接口的FindPin、FindInterface等函数还是相当有用的。

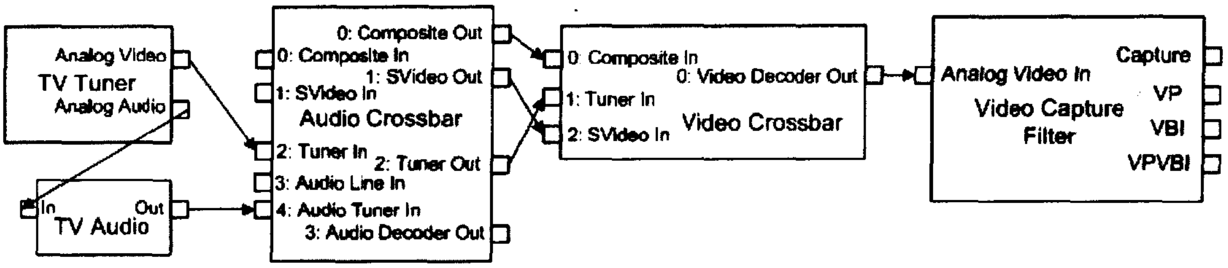


图 2.10　采集设备 Filter 的前端链路

下面列出了CAnalogInputFilters类中完成采集设备输入部分Filter的构建过程，以及用于将输入部分与输出部分相连的接口函数：GetVideoOutPin、GetPreviewPin、GetAudioOutPin等。

```
//
// CAnalogInputFilters.cpp
//
// 输入部分的创建函数
BOOL CAnalogInputFilters::Create(void)
{
    ASSERT(mGraph);
    ASSERT(mLiveCapture);

    HRESULT hr = S_OK;
    // 创建视频采集设备Filter的包装对象
    mVideoCapture = new CVideoCaptureFilter(mGraph->GetGraph(),
        mLiveCapture->GetVideoDevice());
    // 创建音频采集设备Filter的包装对象
    mAudioCapture = new CAudioCaptureFilter(mGraph->GetGraph(),
        mLiveCapture->GetAudioDevice());
    // 创建一个Smart Tee和一个Overlay Mixer
    mSmartTee     = new CSmartTeeFilter(mGraph->GetGraph());
    mOverlayMixer = new COverlayMixerFilter(mGraph->GetGraph());

    BOOL pass = CLiveInputFilters::Create();
    if (pass)
    {
        // 真正创建视频采集设备Filter并加入到Filter Graph中
        pass = mVideoCapture->CreateFilter();
    }
    if (pass)
    {
        // 创建Capture Graph Builder组件
        hr = CoCreateInstance((REFCLSID)CLSID_CaptureGraphBuilder2, 0,
            CLSCTX_INPROC, (REFIID)IID_ICaptureGraphBuilder2, (void
            **)&mGraphBuilder);
        pass = (mGraphBuilder != NULL);
    }
```

```
if (pass)
{
    // 将IGraphBuilder接口设置给Capture Graph Builder组件
    mGraphBuilder->SetFiltergraph(mGraph->GetGraph());
    // 为视频采集设备构建预览用的Filter链路
    hr = mGraphBuilder->RenderStream(&PIN_CATEGORY_PREVIEW,
    &MEDIATYPE_Video,
        mVideoCapture->GetFilter(), 0, 0);
    if (hr == VFW_S_NOPREVIEWPIN)
    {
        // 有些采集设备没有Preview输出Pin, Capture Graph Builder组件
        // 会自动插入一个Smart Tee接到Capture输出Pin后面
        mHasPreviewPin = FALSE;
    }
    else if (hr != S_OK)
    {
        pass = FALSE;
    }
}
if (pass)
{
    // --- 进行一系列参数设置 ---
    // 创建Crossbar的控制对象（用于选择输入端子）
    CreateCrossBar(mVideoCapture->GetFilter());
    // 将采集设备Filter后面连接的所有Filter都删除
    UDsUtils::NukeDownstream(mGraph->GetGraph(),
        mVideoCapture->GetFilter());
    // 设置视频采集设备的输入端子
    SetVideoConnector(mLiveCapture->GetVideoConnector());
    // 设置采集设备的视频解码制式
    SetVideoResolution(mLiveCapture->GetVideoResolution());
    // 调整采集设备输出图像的尺寸
    AdjustOutput();
    // 判断是否有必要连接一个Smart Tee
    CheckSmartTee();
}
// 创建音频输入部分
if (pass)
{
    // 真正创建音频采集设备Filter并加入到Filter Graph中
    mAudioCapture->CreateFilter();
    // 设置音频的输入端子
    if (mLiveCapture->IsAudioConnectorValid())
    {
        mAudioCapture->SetConnector(mLiveCapture->
            GetAudioConnector());
    }
    else
    {
        mLiveCapture->SetNotifying(FALSE);
        mLiveCapture->SetAudioConnector(mAudioCapture->
            GetConnector());
```

```
            mLiveCapture->SetNotifying(TRUE);
        }
    }
    return pass;
}
// 返回输入部分的视频输出Pin（主要用于采集时连接）
IPin * CAnalogInputFilters::GetVideoOutPin(void)
{
    // 如果已经连接了一个Smart Tee，则返回Smart Tee的Capture输出Pin
    if (mSmartTee && mSmartTee->GetFilter())
    {
        return mSmartTee->GetCapturePin();
    }
    // 如果没有连接Smart Tee，则返回采集设备Filter的Capture输出Pin
    return FindVideoPin(&PIN_CATEGORY_CAPTURE);
}
// 返回输入部分的视频预览Pin（主要用于连接）
IPin * CAnalogInputFilters::GetPreviewPin(void)
{
    if (IsAGPCard())  // 如果是AGP设备
    {
        BOOL pass = TRUE;
        // 首先连接一个Overlay Mixer到VP Pin上
        if (!mOverlayMixer->GetFilter())
        {
            mOverlayMixer->CreateFilter();
            IPin * pOut  = FindVideoPin(&PIN_CATEGORY_VIDEOPORT);
            IPin * pIn  = mOverlayMixer->GetPin(TRUE);
            pass = mGraph->ConnectFilters(pOut, pIn);
        }
        // 然后返回Overlay Mixer的输出Pin（用于连接到Video Renderer上）
        if (pass)
        {
            return mOverlayMixer->GetPin(FALSE);
        }
    }

    // 如果不是AGP设备，且已经连接了一个Smart Tee，则返回Smart Tee的Preview输出
    // Pin
    if (mSmartTee && mSmartTee->GetFilter())
    {
        return mSmartTee->GetPreviewPin();
    }
    IPin * pin = NULL;
    if (mHasPreviewPin) // 采集设备是否有Preview输出Pin?
    {
        // 如果采集设备Filter本身有Preview Pin，则返回这个Pin
        pin = FindVideoPin(&PIN_CATEGORY_PREVIEW);
    }
    else
    {
        // 如果采集设备Filter本身没有Preview Pin，则返回Capture Pin
```

```
        pin = FindVideoPin(&PIN_CATEGORY_CAPTURE);
    }
    return pin;
}
// 返回输入部分的音频输出Pin（主要用于连接）
IPin * CAnalogInputFilters::GetAudioOutPin(void)
{
    // 返回音频采集设备Filter的输出Pin
    if (mAudioCapture->GetFilter())
    {
        return mAudioCapture->GetPin(FALSE);
    }
    return NULL;
}
// 判断当前的采集设备是不是AGP接口的，如果采集设备Filter上有VP Pin则是，否则就不是
BOOL CAnalogInputFilters::IsAGPCard(void)
{
    if (FindVideoPin(&PIN_CATEGORY_VIDEOPORT))
    {
        return TRUE;
    }
    return FALSE;
}
// 使用ICaptureGraphBuilder2::FindPin查询得到Filter上的某个Pin
IPin * CAnalogInputFilters::FindVideoPin(const GUID * inCategory)
{
    if (mVideoCapture && mGraphBuilder)
    {
        IPin * pPin = NULL;
        HRESULT hr = mGraphBuilder->FindPin(
                mVideoCapture->GetFilter(),     // 指定Filter
                PINDIR_OUTPUT,                  // 指定Pin的方向
                inCategory,                     // 指定Pin的类型
                NULL,
                FALSE,
                0,
                &pPin
                );
        if (SUCCEEDED(hr))
        {
            pPin->Release();
            return pPin;
        }
    }
    return NULL;
}
// 判断是否有必要连接一个Smart Tee
// 预览模式：不用接Smart Tee
// 采集模式：如果设备不是AGP接口的，则接入一个Smart Tee
void CAnalogInputFilters::CheckSmartTee(void)
{
    // 仅在对非AGP设备进行采集时接入一个Smart Tee
```

```
    if (!IsAGPCard() && mLiveCapture->GetWorkMode() == MD_Capture)
    {
        // 创建Smart Tee Filter并将其加入到Filter Graph中
        BOOL pass = mSmartTee->CreateFilter();
        if (pass && mVideoCapture->GetFilter())
        {
            // 在Capture输出Pin后面接上Smart Tee
            IPin * pOut = FindVideoPin(&PIN_CATEGORY_CAPTURE);
            IPin * pIn  = mSmartTee->GetInputPin();
            pass = mGraph->ConnectFilters(pOut, pIn);
        }
    }
}

//
// UDsUtils.cpp
//
// 这是一个递归调用的函数：给定一个Filter，删除与该Filter连接的下游的所有Filter
void UDsUtils::NukeDownstream(IGraphBuilder * inGraph, IBaseFilter *
    inFilter)
{
    if (!inGraph || !inFilter)
    {
        return ;
    }

    IEnumPins * pinEnum = NULL;
    // 创建一个Pin的枚举器
    if (SUCCEEDED(inFilter->EnumPins(&pinEnum)))
    {
        pinEnum->Reset();

        IPin * pin     = NULL;
        ULONG  fetched = 0;
        // 枚举该Filter上所有的Pin
        while (SUCCEEDED(pinEnum->Next(1, &pin, &fetched)) && fetched)
        {
            if (pin)
            {
                // 得到当前Pin连接对方的Pin指针
                IPin * connectedPin = NULL;
                pin->ConnectedTo(&connectedPin);
                if (connectedPin)
                {
                    // 查询Pin的信息（获取Pin的方向）
                    PIN_INFO pininfo;
                    if (SUCCEEDED(connectedPin->QueryPinInfo(&pininfo)))
                    {
                    pininfo.pFilter->Release();
                    // 如果连接对方是输入Pin（说明当前枚举得到的是输出Pin），则递
                    // 归调用NukeDownstream函数，首先将下一级（乃至再下一级）的
                    // 所有Filter删除
```

```
                    if (pininfo.dir == PINDIR_INPUT)
                    {
                        NukeDownstream(inGraph, pininfo.pFilter);
                        inGraph->Disconnect(connectedPin);
                        inGraph->Disconnect(pin);
                        inGraph->RemoveFilter(pininfo.pFilter);
                    }
                    }
                    connectedPin->Release();
                }
                pin->Release();
            }
        }
        pinEnum->Release();
    }
}
```

AVCap程序设计了3个控制类：CGraphController、CPreviewController和Ccapture Controller。CGraphController是CPreviewController和CCaptureController的父类，主要定义公共接口，以及实现一些预览和采集都要用到的功能函数；CPreviewController主要负责预览用的Filter Graph的构建；CCaptureController主要负责采集用的Filter Graph的构建。CGraphController类的Activate函数体现了AVCap程序创建Filter Graph的通用过程，其中CreateInputFilters是公共的（在CGraphController类中实现），而CreateOutputFilters和ConnectFilters都可以在子类CPreviewController或CCaptureController中定制。

```
//
// CGraphController.h
//

#ifndef __H_CGraphController__
#define __H_CGraphController__

#include "CVideoRendererFilter.h"
#include "CAudioRendererFilter.h"
#include "CMsgReceiver.h"

class CDXGraph;
class CDXFilter;
class CVideoWindow;
class CLiveCapture;
class CLiveInputFilters;
class CGraphController : public CMsgReceiver
{
protected:
    // CDXGraph是Filter Graph的封装类
    CDXGraph *          mGraph;             // Filter Graph
    CVideoWindow *      mVideoWindow;       // 视频显示窗口
    CLiveCapture *      mLiveCapture;       // 配置参数
    BOOL               mIsGraphActive;     // 指示当前Filter Graph是否构建成功

    CLiveInputFilters *     mInputFilters;    // Filter链路的输入部分
```

```
    // CVideoRendererFilter是Video Renderer的封装类
    CVideoRendererFilter *  mVideoRenderer; // Video Renderer
    // CAudioRendererFilter是Audio Renderer的封装类
    CAudioRendererFilter *  mAudioRenderer; // Audio Renderer
public:
    CGraphController();
    virtual ~CGraphController();
    // 设置视频显示窗口
    void SetVideoWindow(CVideoWindow * inWindow);
    // 设置配置参数对象
    void SetLiveCapture(CLiveCapture * inData);
    // 得到Filter链路输入部分的包装对象指针
    CLiveInputFilters * GetInputFilters(void) { return mInputFilters;}
    // 判断当前设备是否丢失（热插拔时使用）
    virtual BOOL IsActiveDeviceLost(IUnknown * pLost);
    // 当前播放时刻抓帧到一个位图文件
    virtual BOOL SnapshotToFile(const char * inFile);
    // 构建Filter Graph
    BOOL Activate(void);
    // 销毁Filter Graph
    void Deactive(void);
    // 判断输入视频信号是不是PAL制式
    virtual BOOL IsPAL(void);
    // Filter Graph的控制：播放、暂停、停止
    BOOL PlayGraph(void);
    BOOL PauseGraph(void);
    BOOL StopGraph(void);
    // 判断当前Filter Graph是否构建成功
    BOOL IsGraphActive(void);
    // 响应自定义消息
    virtual bool ReceiveMessage(MessageT inMessage, void * ioParam,
        void * ioParam2);
protected:
    // 创建、销毁Filter Graph
    BOOL CreateGraph(void);
    void DeleteGraph(void);
    // 连接、断开输入部分与输出部分的Filter（虚函数，可供子类定制）
    virtual BOOL ConnectFilters(void);
    virtual void DisconnectFilters(void);
    // 创建、销毁输入部分的Filter（虚函数，可供子类定制）
    virtual BOOL CreateInputFilters(void);
    virtual void ReleaseInputFilters(void);
    // 创建、销毁输出部分的Filter（虚函数，可供子类定制）
    virtual BOOL CreateOutputFilters(void);
    virtual void ReleaseOutputFilters(void);
};

#endif // __H_CGraphController__

//
// CGraphController.cpp
//
```

```
// 构建一个完整的Filter Graph，包括必要的Filter的创建、连接等
BOOL CGraphController::Activate(void)
{
    // 创建Filter Graph
    BOOL pass = CreateGraph();
    if (pass)
    {
        // 创建输入部分的Filter
        pass = CreateInputFilters();
    }
    if (pass)
    {
        // 创建输出部分的Filter（子类可以定制）
        pass = CreateOutputFilters();
    }
    if (pass)
    {
        // 将输入部分的Filter与输出部分的Filter相连（子类可以定制）
        pass = ConnectFilters();
    }
    if (pass && mVideoWindow)
    {
        // 设置视频窗口，并自动播放Filter Graph
        mVideoWindow->SetGraph(mGraph);
        PlayGraph();
    }
    mIsGraphActive = pass;
    return pass;
}
// 创建Filter Graph
BOOL CGraphController::CreateGraph(void)
{
    mGraph = new CDXGraph();
    if (mGraph->Create())
    {
        mGraph->AddMsgReceiver(this);
        return TRUE;
    }
    DeleteGraph();
    return FALSE;
}
// 销毁Filter Graph
void CGraphController::DeleteGraph(void)
{
    if (mGraph)
    {
        mGraph->RemoveMsgReceiver(this);
        delete mGraph;
        mGraph = NULL;
    }
}
// 创建输入部分的Filter（预览和采集的公共部分）
```

```
BOOL CGraphController::CreateInputFilters(void)
{
    if (mLiveCapture)
    {
        ReleaseInputFilters();

        // 根据设备的类型，分别实例化不同的采集设备输入包装类
        switch (mLiveCapture->GetDeviceType())
        {
        case DT_DV:
            mInputFilters = new CDVInputFilters(mGraph);
            break;

        case DT_Analog_WDM:
        case DT_Analog_VFW:
        default:
            // 注意：CTVInputFilters 是CAnalogInputFilters的子类
            mInputFilters = new CTVInputFilters(mGraph);
            break;
        }

        // 设置参数对象
        mInputFilters->SetLiveCapture(mLiveCapture);
        // 创建采集设备的前端部分
        return mInputFilters->Create();
    }
    return FALSE;
}
// 销毁输入部分的Filter
// #define SAFE_DELETE(p)  { if(p) { delete (p); (p)=NULL; } }
void CGraphController::ReleaseInputFilters(void)
{
    SAFE_DELETE(mInputFilters);
}
// 创建输出部分的Filter。因为预览和采集都要用到Video Renderer和Audio Renderer,
// 因此在父类中创建这两个Filter
BOOL CGraphController::CreateOutputFilters(void)
{
    ReleaseOutputFilters();

    BOOL pass = TRUE;
    // 创建Video Renderer
    mVideoRenderer = new CVideoRendererFilter(mGraph->GetGraph());
    // 设置是否优先使用VMR的标记
    mVideoRenderer->SetVMRPreferred(mLiveCapture->IsVMRPreferred());
    pass = mVideoRenderer->CreateFilter();
    // 创建Audio Renderer
    mAudioRenderer = new CAudioRendererFilter(mGraph->GetGraph());
    pass = mAudioRenderer->CreateFilter();
    return pass;
}
// 销毁输出部分的Filter
```

```
void CGraphController::ReleaseOutputFilters(void)
{
    SAFE_DELETE(mVideoRenderer);
    SAFE_DELETE(mAudioRenderer);
}
// 连接输入和输出部分的Filter（留给子类去实现）
BOOL CGraphController::ConnectFilters(void)
{
    return FALSE;
}

//
// CPreviewController.cpp
// CPreviewController类负责预览用的Filter Graph的构建
// 重写虚函数ConnectFilters，完成预览用的输入部分与输出部分的Filter连接
BOOL CPreviewController::ConnectFilters(void)
{
    BOOL pass   = FALSE;
    IPin * pOut = NULL;
    IPin * pIn  = NULL;

    // 判断是否需要连接一个即时抓图的Filter
    if (mGrabber && mGrabber->GetFilter())
    {
        // 如果需要即时抓图，则首先连接一个Image Grabber Filter，然后将Image
        // Grabber的输出Pin连接到Video Renderer上主要的实现代码省略（参见2.2.9
        // 节）
        // ……
        pOut = mGrabber->GetPin(FALSE);
    }
    else
    {
        // 如果不需要即时抓图，则直接将Preview Pin连接到Video Renderer
        pOut = mInputFilters->GetPreviewPin();
    }
    // 最后将视频链路连接到Video Renderer上
    pIn = mVideoRenderer->GetInputPin();
    if (pOut && pIn)
    {
        pass = mGraph->ConnectFilters(pOut, pIn);
    }

    // 连接输入部分的音频输出Pin到Audio Renderer上
    pOut = mInputFilters->GetAudioOutPin();
    pIn  = mAudioRenderer->GetPin(TRUE);
    if (pass && pOut && pIn)
    {
        pass = mGraph->ConnectFilters(pOut, pIn);
    }
    return pass;
}
// 重写虚函数CreateOutputFilters，完成预览用的输出部分Filter的创建
```

```
BOOL CPreviewController::CreateOutputFilters(void)
{
    // 首先调用父类的CreateOutputFilters函数创建Video Renderer和Audio
    // Renderer
    BOOL pass = CGraphController::CreateOutputFilters();
    if (pass)
    {
        // 如果允许即时抓图，则创建一个Image Grabber Filter
        // 是否允许即时抓图通过参数对象中的一个标记来控制
        if (mLiveCapture->IsGrabEnabled())
        {
            // 创建Image Grabber Filter并将其加入Filter Graph
            mGrabber = new CImageGrabberFilter(mGraph->GetGraph());
            pass = mGrabber->CreateFilter();
            // 采集输出是场图像还是帧图像？如果是场图像，Image Grabber Filter内部
            // 需要进行一些处理
            mGrabber->SetFieldPicture(mInputFilters->IsFieldPicture());
        }
    }
    return pass;
}
// 重写虚函数ReleaseOutputFilters，销毁输出部分的Filter
void CPreviewController::ReleaseOutputFilters(void)
{
    // 销毁CPreviewController类中创建的Filter
    SAFE_DELETE(mGrabber);
    SAFE_DELETE(mAviDecoder);
    SAFE_DELETE(mNullRenderer);
    // 销毁父类中创建的Filter
    CGraphController::ReleaseOutputFilters();
}

//
// CCaptureController.cpp
// CCaptureController类负责采集用的Filter Graph的构建
// 重写虚函数ConnectFilters，完成采集用的输入部分与输出部分的Filter连接
BOOL CCaptureController::ConnectFilters(void)
{
BOOL pass   = FALSE;
    // 将输入部分与输入部分的Filter相连（代码省略）
    // ……
    return pass;
}
// 重写虚函数CreateOutputFilters，完成采集用的输出部分Filter的创建
BOOL CCaptureController::CreateOutputFilters(void)
{
    // 首先调用父类的CreateOutputFilters函数创建Video Renderer和Audio
    // Renderer
    BOOL pass = CGraphController::CreateOutputFilters();
    if (pass)
    {
        // 创建编码、合成、写文件需要的Filter（代码省略）
```

```
        // ......
    }
    return pass;
}
// 重写虚函数ReleaseOutputFilters，销毁输出部分的Filter
void CCaptureController::ReleaseOutputFilters(void)
{
    // 销毁CCaptureController类中创建的Filter（代码省略）
    // ......
    // 销毁父类中创建的Filter
    CGraphController::ReleaseOutputFilters();
}
```

从上面的代码中可以看出，AVCap程序在具体实现时使用了很多自定义类，如CDXGraph（封装了Filter Graph的控制功能）、CDXFilter（封装了DirectShow Filter的相关操作，并成为其他Filter包装类的父类）、CVideoWindow（用于视频显示的窗口类）、CLiveCapture（配置参数类）、CLiveInputFilters（输入部分Filter的基类）、CDVInputFilters（负责数码摄像机输入）、CAnalogInputFilters（负责普通的采集设备输入）、CTVInputFilters（负责带有电视接收功能的采集设备输入）等，这些类的功能说明及相互关系将在2.4.2节中再次介绍。

顺便提一下音频的预览。AVCap程序构建了如图2.11所示的Filter Graph：将音频采集设备的Capture输出Pin连接到Audio Renderer上。实际上，有些采集设备会有一个音频的输出端口，将这个端口与声卡的Line In端口用接口线相连。只要Filter Graph运行起来就能听到声音，未必要构建如图2.11所示的Filter Graph。

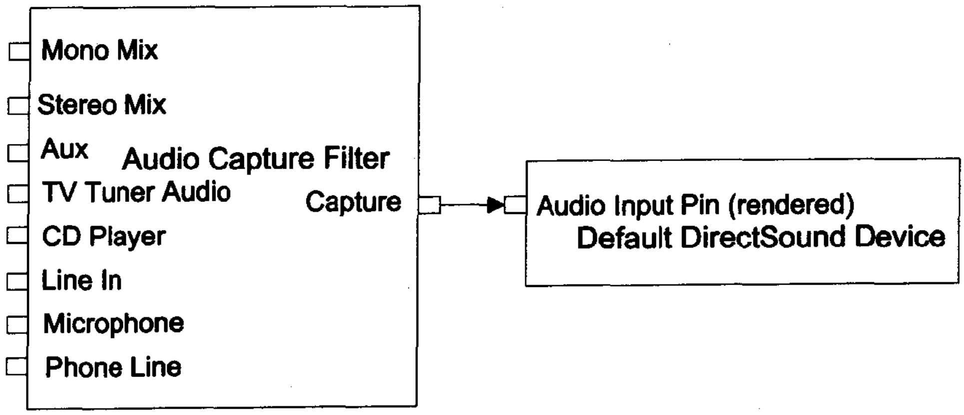


图 2.11　音频预览用的 Filter Graph

提示：音频采集设备的各个输入Pin不需要连接到任何其他的Filter上。这些Pin仅仅是音频采集设备（也就是声卡）各个输入端子的象征性表示。需要对某个输入端子进行音频的预览（或采集）时，在确认这个输入端子的物理连接正常的前提下，只需在程序中通过Pin上的IAMAudioInputMixer接口进行控制就可以了。

2.2.4 使用高效的VMR

传统的Video Renderer一般情况下使用GDI函数进行视频图像的显示，当显卡能够支持DirectDraw时则优先（但不总是）使用DirectDraw API函数。因此，有时候Video Renderer的工作效率并不高。另外，Video Renderer在最初设计的时候就只有一个输入Pin，因此它不支持多个视频流的合成。传统的Video Renderer已经越来越不能满足实际应用的需要。于是，DirectShow发布了它的替代品VMR (Video Mixing Renderer)。

VMR不再使用GDI函数进行视频图像的显示。目前，VMR有两个版本：VMR-7和VMR-9。VMR-7仅在Windows XP下可用，Filter内部采用了DirectDraw 7的技术，是Windows XP下默认的用于视频显示的Renderer（替代了传统的Video Renderer）；VMR-9适合于任何支持DirectX的操作系统（任何安装了DirectX 9.0的操作系统上都能使用这个Filter），Filter内部采用了Direct3D 9的技术，但任何时候都不是默认的用于视频显示的Renderer。（所谓“默认的”Video Renderer，其实就是指使用智能连接方法构建含有视频分支的Filter Graph时，能够被自动加入并连接的Video Renderer。）

提示：VMR的性能跟显卡的性能直接相关。因此在使用VMR之前，请确认你的显卡性能足够出色，或者升级你的显卡驱动程序。

AVCap程序使用了一个包装类CVideoRendererFilter，来集中控制到底使用哪一个Video Renderer。CVideoRendererFilter类中定义了一个标记变量mVMRPreferred，它决定了3个Renderer（传统的Video Renderer、VMR-7和VMR-9）使用的优先级。

```
//
// CVideoRendererFilter.h
//

#ifndef __H_CVideoRendererFilter__
#define __H_CVideoRendererFilter__

#include "CDXFilter.h"

class CVideoRendererFilter : public CDXFilter
{
private:
    BOOL    mVMRPreferred;  // 标记变量，指示是否优先使用VMR
public:
    CVideoRendererFilter(IGraphBuilder * inGraph);
    virtual ~CVideoRendererFilter(void);
    // 重写Filter的创建函数（实现VMR使用的优先策略控制）
    virtual BOOL CreateFilter(void);
    // 设置是否优先使用VMR的标记
    void SetVMRPreferred(BOOL inPreferred);
    // 得到Video Renderer的输入Pin
    IPin * GetInputPin(void);
};

#endif // __H_CVideoRendererFilter__
```

```
//
// CVideoRendererFilter.cpp
//

#include "stdafx.h"
#include "CVideoRendererFilter.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

// VMR-9的CLSID
GUID CLSID_VMR9 = {
0x51B4ABF3, 0x748F, 0x4E3B, 0xA2, 0x76, 0xC8, 0x28, 0x33, 0x0E, 0x92, 0x6A};

// VMR-7的CLSID
GUID CLSID_VMR7 = {
0x6BC1CFFA, 0x8FC1, 0x4261, 0xAC, 0x22, 0xCF, 0xB4, 0xCC, 0x38, 0xDB, 0x50};

//////////////////////////////////////////////////////////////////////////////
CVideoRendererFilter::CVideoRendererFilter(IGraphBuilder * inGraph) :
CDXFilter(inGraph, CLSID_VideoRenderer, "Video Renderer")
{
    // 默认情况下仍然使用传统的Video Renderer
    mVMRPreferred = FALSE;
}
CVideoRendererFilter::~CVideoRendererFilter(void)
{
}
// 创建Filter，并将它加入到Filter Graph中
BOOL CVideoRendererFilter::CreateFilter(void)
{
    if (!mVMRPreferred)
    {
        // 如果不优先使用VMR，则创建传统的Video Renderer
        mClsid = CLSID_VideoRenderer;
        return CDXFilter::CreateFilter();
    }
    else
    {
        // 优先使用VMR-9
        mClsid = CLSID_VMR9;
        BOOL pass = CDXFilter::CreateFilter();
        if (!pass)
        {
            // 如果VMR-9不能创建，则创建VMR-7
            mClsid = CLSID_VMR7; // Only available on XP
            pass = CDXFilter::CreateFilter();
        }
```

```
            if (!pass)
            {
                // 如果VMR-9和VMR-7都不能创建，则仍然创建传统的Video Renderer
                mClsid = CLSID_VideoRenderer;
                pass = CDXFilter::CreateFilter();
            }
            return pass;
        }
    }
    // 设置是否优先使用VMR的标记
    void CVideoRendererFilter::SetVMRPreferred(BOOL inPreferred)
    {
        mVMRPreferred = inPreferred;
    }
    // 得到Video Renderer的输入Pin（根据Pin的名字查找）
    IPin * CVideoRendererFilter::GetInputPin(void)
    {
        if (mClsid == CLSID_VideoRenderer)
        {
            // 传统的Video Renderer的输入Pin名字为Input
            return CDXFilter::GetPin(TRUE, "Input");
        }
        else
        {
            // VMR的（第一个）输入Pin的名字为VMR Input0
            return CDXFilter::GetPin(TRUE, "VMR Input0");
        }
    }
```

由于VMR连接的媒体类型通常是YUY2、UYVY等YUV格式，因此在采集设备输入部分的构建过程中需要进行一些修正：从Preview（或Capture）输出Pin上支持的所有媒体类型中选出一种YUV格式，然后再进行输出图像大小、帧率等参数的设置。（请读者重点查看_Trying_VMR宏包裹的代码块。）

```
    //
    // CAnalogInputFilters.cpp
    //
    #define _Trying_VMR  // 定义一个使用VMR的宏
    BOOL CAnalogInputFilters::Create(void)
    {
        // 初始化代码（省略）
        // ……

        BOOL pass = CLiveInputFilters::Create();
        if (pass)
        {
            // 创建视频输入部分（省略）
            // ……
        }
        if (pass)
        {
            // 进行一系列参数设置（省略）
```

```
        // ......

#ifdef _Trying_VMR // 如果定义了这个宏，则执行下列代码
        if (!IsAGPCard() && mLiveCapture->GetWorkMode() == MD_Preview)
        {
            // 因为AGP设备实现预览的效率已经很高了，所以仅在非AGP设备、预览模式下
            // 优先使用VMR
            mLiveCapture->SetVMRPreferred(TRUE);
            // 在使用VMR的情况下，可以直接使用IBasicVideo::GetCurrentImage
            // 进行即时抓图，而不用额外连接一个Image Grabber Filter
            mLiveCapture->SetImageGrab(FALSE);
        }
        else
        {
            // AGP设备或采集模式下不使用VMR
            mLiveCapture->SetVMRPreferred(FALSE);
            // 进行即时抓图需要连接一个Image Grabber Filter
            mLiveCapture->SetImageGrab(TRUE);
        }
#endif // _Trying_VMR
        // 调整输出图像的尺寸、帧率等参数
        AdjustOutput();
        CheckSmartTee();
    }
    if (pass)
    {
        // 创建音频输入部分（省略）
        // ......
    }
    return pass;
}
// 调整Preview Pin和Capture Pin输出的媒体类型，包括输出图像尺寸、帧率等
void CAnalogInputFilters::AdjustOutput(void)
{
    HRESULT hr        = S_OK;
    AM_MEDIA_TYPE * pmt = NULL;

    // 如果优先使用VMR，则为VMR挑选一种YUV的媒体类型
#ifdef _Trying_VMR
    if (mLiveCapture->IsVMRPreferred())
    {
        pmt = SelectMediaTypeForVMR();
    }
#endif // _Trying_VMR
    if (!pmt)
    {
        // 如果不优先使用VMR，则优先挑选一种RGB的媒体类型
        pmt = SelectMediaType();
    }
    if (pmt)
    {
        // 设置图像尺寸、帧率等参数（省略）
```

```
        // ......
        DeleteMediaType(pmt);
    }
}
// 为VMR挑选一种YUV的媒体类型：YUY2或者UYVY
AM_MEDIA_TYPE * CAnalogInputFilters::SelectMediaTypeForVMR(void)
{
    VIDEO_STREAM_CONFIG_CAPS  pSCC;
    AM_MEDIA_TYPE * pmt = NULL;
    HRESULT hr = S_OK;
    int nCounts, nSize;
    int preferredIndex = -1;
    // 定义几种媒体类型的优先顺序
    enum { YUY2 = 0, UYVY, Unknown } currentPreferred, temp;
    currentPreferred = Unknown;
    // 获得Capture输出Pin上的IAMStreamConfig接口
    IAMStreamConfig * pCfg = GetStreamConfig(&PIN_CATEGORY_CAPTURE);
    // 得到Capture Pin上支持的采集输出格式的个数
    pCfg->GetNumberOfCapabilities(&nCounts, &nSize);
    // 遍历Capture输出Pin上所有支持的媒体类型
    for (int i = 0; i < nCounts; i++)
    {
        if (pCfg->GetStreamCaps(i, &pmt, (BYTE *)&pSCC) == S_OK)
        {
            if (pmt->subtype == MEDIASUBTYPE_YUY2)
            {
                temp = YUY2;
            }
            else if (pmt->subtype == MEDIASUBTYPE_UYVY)
            {
                temp = UYVY;
            }
            else
            {
                temp = Unknown;
            }
            // 保存一种更优先的格式
            if (temp < currentPreferred)
            {
                currentPreferred = temp;
                preferredIndex   = i;
            }
            DeleteMediaType(pmt);
        }
    }
    if (preferredIndex != -1)
    {
        // 获取优先的YUV媒体类型
        hr = pCfg->GetStreamCaps(preferredIndex, &pmt, (BYTE *)&pSCC);
    }
    return pmt;
}
```

```
// 优先挑选出一种RGB的媒体类型
// 优先顺序为：RGB32, RGB24, RGB565, RGB555, YUY2, UYVY
AM_MEDIA_TYPE * CAnalogInputFilters::SelectMediaType(void)
{
    VIDEO_STREAM_CONFIG_CAPS  pSCC;
    AM_MEDIA_TYPE * pmt = NULL;
    HRESULT hr = S_OK;
    int nCounts, nSize;
    int preferredIndex = -1;
    // 定义几种媒体类型的优先顺序
    enum {
        RGB32 = 0, RGB24, RGB565, RGB555, YUY2, UYVY, Unknown
    } currentPreferred, temp;
    currentPreferred = Unknown;
    // 获得Capture输出Pin上的IAMStreamConfig接口
    IAMStreamConfig * pCfg = GetStreamConfig(&PIN_CATEGORY_CAPTURE);
    // 得到Capture Pin上支持的采集输出格式的个数
    pCfg->GetNumberOfCapabilities(&nCounts, &nSize);
    // 遍历Capture输出Pin上支持的所有媒体类型
    for (int i = 0; i < nCounts; i++)
    {
        if (pCfg->GetStreamCaps(i, &pmt, (BYTE *)&pSCC) == S_OK)
        {
            if (pmt->subtype == MEDIASUBTYPE_RGB32)
            {
                temp = RGB32;
            }
            else if (pmt->subtype == MEDIASUBTYPE_RGB24)
            {
                temp = RGB24;
            }
            else if (pmt->subtype == MEDIASUBTYPE_RGB565)
            {
                temp = RGB565;
            }
            else if (pmt->subtype == MEDIASUBTYPE_RGB555)
            {
                temp = RGB555;
            }
            else if (pmt->subtype == MEDIASUBTYPE_YUY2)
            {
                temp = YUY2;
            }
            else if (pmt->subtype == MEDIASUBTYPE_UYVY)
            {
                temp = UYVY;
            }
            else
            {
                temp = Unknown;
            }
            // 保存一种更优先的格式
```

```
                if (temp < currentPreferred)
                {
                    currentPreferred = temp;
                    preferredIndex   = i;
                }
                DeleteMediaType(pmt);
            }
        }
        if (preferredIndex != -1)
        {
            // 获取优先的RGB媒体类型
            hr = pCfg->GetStreamCaps(preferredIndex, &pmt, (BYTE *)&pSCC);
        }
        else
        {
            // 如果没有得到已知的优先使用的媒体类型,
            // 则获取Capture输出Pin上当前使用的媒体类型
            hr = pCfg->GetFormat(&pmt);
        }
        return pmt;
}
// 获取指定Pin上的IAMStreamConfig接口
IAMStreamConfig * CAnalogInputFilters::GetStreamConfig(const GUID *
        inCategory)
{
    IAMStreamConfig * pConfig = NULL;
    if (mGraphBuilder != NULL)
    {
        mGraphBuilder->FindInterface(
            inCategory,                     // 指定Pin的类型
            &MEDIATYPE_Video,               // 指定需要的是一个视频相关的Pin
            mVideoCapture->GetFilter(),     // 指定Pin所属的Filter
            IID_IAMStreamConfig,            // 查询的接口GUID
            (void **)&pConfig
            );
    }
    if (pConfig)
    {
        pConfig->Release();
    }
    return pConfig;
}
```

小知识：RGB与YUV

计算机彩色显示器显示色彩的原理与彩色电视机一样，都是采用R（Red）、G（Green）、B（Blue）相加混色的原理：通过发射出3种不同强度的电子束，使屏幕内侧覆盖的红、绿、蓝磷光材料发光而产生色彩。这种色彩的表示方法称为RGB色彩空间表示（它也是多媒体计算机技术中用得最多的一种色彩空间表示方法）。

根据三基色原理，任意一种色光F都可以用不同分量的R、G、B三色相加混合而成。

$$F=r[R]+g[G]+b[B]$$

其中，r、g、b分别为三基色参与混合的系数。当三基色分量都为0（最弱）时混合为黑色光；而当三基色分量都为k（最强）时混合为白色光。调整r、g、b 3个系数的值，可以混合出介于黑色光和白色光之间的各种各样的色光。

那么YUV又从何而来呢？在现代彩色电视系统中，通常采用三管彩色摄像机或彩色CCD摄像机进行摄像，然后把摄得的彩色图像信号经分色、分别放大校正后得到RGB，再经过矩阵变换电路得到亮度信号Y和两个色差信号R－Y（即U）、B－Y（即V），最后发送端将亮度和色差3个信号分别进行编码，用同一信道发送出去。这种色彩的表示方法就是所谓的YUV色彩空间表示。

采用YUV色彩空间的重要性是它的亮度信号Y和色差信号U、V是分离的。如果只有Y信号分量而没有U、V分量，那么这样表示的图像就是黑白灰度图像。彩色电视采用YUV空间正是为了用亮度信号Y解决彩色电视机与黑白电视机的兼容问题，使黑白电视机也能接收彩色电视信号。

YUV与RGB相互转换的公式如下（RGB取值范围均为0~255）：

$$Y = 0.299R + 0.587G + 0.114B$$
$$U = -0.147R - 0.289G + 0.436B$$
$$V = 0.615R - 0.515G - 0.100B$$

$$R = Y + 1.14V$$
$$G = Y - 0.39U - 0.58V$$
$$B = Y + 2.03U$$

在DirectShow中，常见的RGB格式有RGB1、RGB4、RGB8、RGB565、RGB555、RGB24、RGB32、ARGB32等；常见的YUV格式有YUY2、YUYV、YVYU、UYVY、AYUV、Y41P、Y411、Y211、IF09、IYUV、YV12、YVU9、YUV411、YUV420等。作为视频媒体类型的辅助说明类型（Subtype），它们对应的GUID如表2.3所示。

表2.3　常见的RGB和YUV格式

GUID	格式描述
MEDIASUBTYPE_RGB1	2色，每个像素用1位表示，需要调色板
MEDIASUBTYPE_RGB4	16色，每个像素用4位表示，需要调色板
MEDIASUBTYPE_RGB8	256色，每个像素用8位表示，需要调色板
MEDIASUBTYPE_RGB565	每个像素用16位表示，RGB分量分别使用5位、6位、5位
MEDIASUBTYPE_RGB555	每个像素用16位表示，RGB分量都使用5位（剩下的1位不用）
MEDIASUBTYPE_RGB24	每个像素用24位表示，RGB分量各使用8位
MEDIASUBTYPE_RGB32	每个像素用32位表示，RGB分量各使用8位（剩下的8位不用）
MEDIASUBTYPE_ARGB32	每个像素用32位表示，RGB分量各使用8位（剩下的8位用于表示Alpha通道值）
MEDIASUBTYPE_YUY2	YUY2格式，以4:2:2方式打包
MEDIASUBTYPE_YUYV	YUYV格式（实际格式与YUY2相同）
MEDIASUBTYPE_YVYU	YVYU格式，以4:2:2方式打包

（续表）

GUID	格式描述
MEDIASUBTYPE_UYVY	UYVY格式，以4:2:2方式打包
MEDIASUBTYPE_AYUV	带Alpha通道的4:4:4 YUV格式
MEDIASUBTYPE_Y41P	Y41P格式，以4:1:1方式打包
MEDIASUBTYPE_Y411	Y411格式（实际格式与Y41P相同）
MEDIASUBTYPE_Y211	Y211格式
MEDIASUBTYPE_IF09	IF09格式
MEDIASUBTYPE_IYUV	IYUV格式
MEDIASUBTYPE_YV12	YV12格式
MEDIASUBTYPE_YVU9	YVU9格式

下面分别介绍各种RGB格式。

- RGB1、RGB4、RGB8都是调色板类型的RGB格式，在描述这些媒体类型的格式细节时，通常会在BITMAPINFOHEADER数据结构后面跟着一个调色板（定义一系列颜色）。它们的图像数据并不是真正的颜色值，而是当前像素颜色值在调色板中的索引。以RGB1（2色位图）为例，比如它的调色板中定义的两种颜色值依次为0x000000（黑色）和0xFFFFFF（白色），那么图像数据001101010111...（每个像素用1位表示）表示对应各像素的颜色为：黑黑白白黑白黑白黑白白白....。
- RGB565使用16位表示一个像素，这16位中的5位用于R，6位用于G，5位用于B。程序中通常使用一个字（WORD，一个字等于两个字节）来操作一个像素。当读出一个像素后，这个字的各个位意义如下：

高字节	低字节
RRRRRGGG	GGGBBBBB

可以组合使用屏蔽字和移位操作来得到RGB各分量的值：

```
#define RGB565_MASK_RED    0xF800
#define RGB565_MASK_GREEN  0x07E0
#define RGB565_MASK_BLUE   0x001F
R = (wPixel & RGB565_MASK_RED) >> 11;      // 取值范围0~31
G = (wPixel & RGB565_MASK_GREEN) >> 5;     // 取值范围0~63
B =  wPixel & RGB565_MASK_BLUE;            // 取值范围0~31
```

- RGB555是另一种16位的RGB格式，RGB分量都用5位表示（剩下的1位不用）。使用一个字读出一个像素后，这个字的各个位意义如下：

高字节	低字节	
XRRRRRGG	GGGBBBBB	（X表示不用，可以忽略）

可以组合使用屏蔽字和移位操作来得到RGB各分量的值：

```
#define RGB555_MASK_RED    0x7C00
#define RGB555_MASK_GREEN  0x03E0
#define RGB555_MASK_BLUE   0x001F
```

```
R = (wPixel & RGB555_MASK_RED) >> 10;       // 取值范围0~31
G = (wPixel & RGB555_MASK_GREEN) >> 5;      // 取值范围0~31
B =  wPixel & RGB555_MASK_BLUE;             // 取值范围0~31
```

- RGB24使用24位来表示一个像素，RGB分量都用8位表示，取值范围为0~255。注意在内存中RGB各分量的排列顺序为：BGR BGR BGR…。通常可以使用RGBTRIPLE数据结构来操作一个像素，它的定义为：

```
typedef struct tagRGBTRIPLE {
  BYTE rgbtBlue;     // 蓝色分量
  BYTE rgbtGreen;    // 绿色分量
  BYTE rgbtRed;      // 红色分量
} RGBTRIPLE;
```

- RGB32使用32位来表示一个像素，RGB分量各用去8位，剩下的8位用作Alpha通道或者不用。（ARGB32就是带Alpha通道的RGB32。）注意在内存中RGB各分量的排列顺序为：BGRA BGRA BGRA…。通常可以使用RGBQUAD数据结构来操作一个像素，它的定义为：

```
typedef struct tagRGBQUAD {
  BYTE     rgbBlue;         // 蓝色分量
  BYTE     rgbGreen;        // 绿色分量
  BYTE     rgbRed;          // 红色分量
  BYTE     rgbReserved;     // 保留字节（用作Alpha通道或忽略）
} RGBQUAD;
```

下面介绍各种YUV格式。YUV格式通常有两大类：打包（packed）格式和平面（planar）格式。前者将YUV分量存放在同一个数组中，通常是几个相邻的像素组成一个宏像素（macro-pixel）；而后者使用3个数组分开存放YUV 3个分量，就像是一个三维平面一样。表2.3中的YUY2到Y211都是打包格式，而IF09到YVU9都是平面格式。（注意：在介绍各种具体格式时，YUV各分量都会带有数字索引，如Y0、U0、V0表示第一个像素的YUV分量，Y1、U1、V1表示第二个像素的YUV分量，依此类推。）

- YUY2（和YUYV）格式为每个像素保留Y分量，而UV分量在水平方向上每两个像素采样一次。一个宏像素为4个字节，实际表示2个像素。（4:2:2的意思为一个宏像素中有4个Y分量、2个U分量和2个V分量。）图像数据中YUV分量排列顺序如下：

  Y0 U0 Y1 V0　　Y2 U2 Y3 V2 …

- YVYU格式与YUY2类似，只是图像数据中YUV分量的排列顺序有所不同：

  Y0 V0 Y1 U0　　Y2 V2 Y3 U2 …

- UYVY格式与YUY2类似，只是图像数据中YUV分量的排列顺序有所不同：

  U0 Y0 V0 Y1　　U2 Y2 V2 Y3 …

- AYUV格式带有一个Alpha通道，并且为每个像素都提取YUV分量，图像数据格式如下：

A0 Y0 U0 V0　　A1 Y1 U1 V1 …

- Y41P（和Y411）格式为每个像素保留Y分量，而UV分量在水平方向上每4个像素采样一次。一个宏像素为12个字节，实际表示8个像素。图像数据中YUV分量排列顺序如下:

U0 Y0 V0 Y1　　U4 Y2 V4 Y3　　Y4 Y5 Y6 Y8 …

- Y211格式在水平方向上Y分量每2个像素采样一次，而UV分量每4个像素采样一次。一个宏像素为4个字节，实际表示4个像素。图像数据中YUV分量排列顺序如下:

Y0 U0 Y2 V0　　Y4 U4 Y6 V4 …

- YVU9格式为每个像素都提取Y分量，而在UV分量的提取时，首先将图像分成若干个4 x 4的宏块，然后每个宏块提取一个U分量和一个V分量。图像数据存储时，首先是整幅图像的Y分量数组，然后就跟着U分量数组，以及V分量数组。IF09格式与YVU9类似。
- IYUV格式为每个像素都提取Y分量，而在UV分量的提取时，首先将图像分成若干个2 x 2的宏块，然后每个宏块提取一个U分量和一个V分量。YV12格式与IYUV类似。
- YUV411、YUV420格式多见于DV数据中，前者用于NTSC制，后者用于PAL制。YUV411为每个像素都提取Y分量，而UV分量在水平方向上每4个像素采样一次。YUV420并非V分量采样为0，而是跟YUV411相比，在水平方向上提高一倍色差采样频率，在垂直方向上以U/V间隔的方式减小一半色差采样，如图2.12所示。

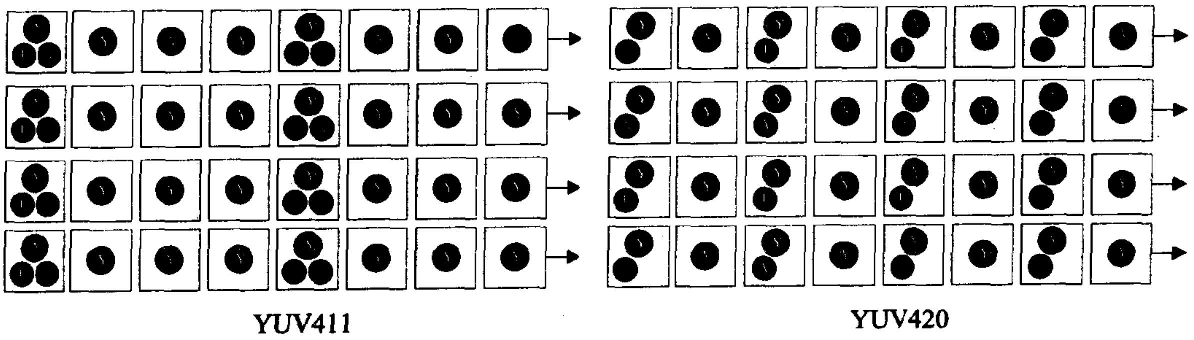


图 2.12　YUV411 和 YUV420 的采样格式

2.2.5　选择输入端子

通常采集设备都有多个输入接口，如AV复合端子、SVideo端子、电视接收等。当物理连接已经就绪的情况下，如何在程序中进行输入端子的选择呢？答案是通过采集设备Filter前端的Crossbar Filter。

采集设备Filter前端一般都会使用一个Crossbar Filter，如图2.13所示。也有些采集设备使用两个Crossbar Filter，如图2.10所示。输入端子的选择实际上就是Crossbar Filter内部对

某个输入Pin与相应的输出Pin进行“导通”的工作(调用Crossbar Filter上实现的IAMCrossbar接口的Route函数),例如要求选择SVideo端子输入,则应将Video SVideo In与Video Decoder Out两个Pin导通(一般需要同时导通Audio Line In与Audio Decoder Out),如果选择电视接收,则应将Video Tuner In与Video Decoder Out两个Pin导通(同时导通Audio Tuner In与Audio Decoder Out),如果选择AV复合端子输入,则应将Video Composite In与Video Decoder Out两个Pin导通(一般需要同时导通Audio Line In与Audio Decoder Out)。

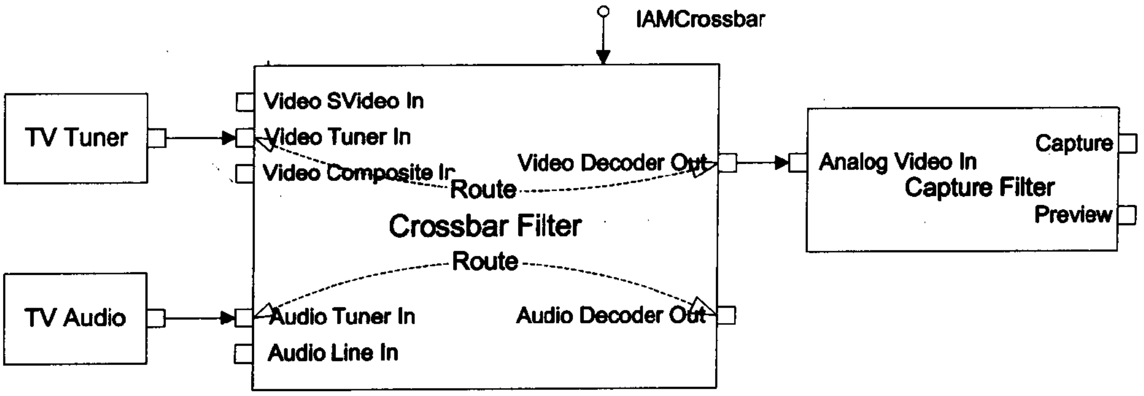


图 2.13 使用一个 Crossbar 的采集设备

操作Crossbar的方法有很多种,但DirectShow SDK中的例子AMCap(位于SDK安装目录的Samples\C++\DirectShow\Capture\AMCap下)提供了一种更为通用的做法:它设计了一个类CCrossbar来建立各个输入端子的导通映射信息,并且提供了非常方便的获取和选择输入端子的接口函数。CCrossbar类在Crossbar.h和Crossbar.cpp两个文件中定义、实现。AVCap程序直接将这两个文件复制到本项目中加以使用。CCrossbar类的定义如下:

```
//
// File: Crossbar.h
//
#ifndef __Crossbar_h__
#define __Crossbar_h__

class CCrossbar
{
private:
    IPin               *m_pStartingPin;
    CRouting            m_RoutingRoot;
    CRoutingList       *m_RoutingList;
    int                m_CurrentRoutingIndex;

    HRESULT BuildRoutingList (
                IPin     *pStartingInputPin,
                CRouting *pCRouting,
                int       Depth);
    HRESULT SaveRouting (CRouting *pRoutingNew);
    HRESULT DestroyRoutingList();
    BOOL    StringFromPinType (TCHAR *pc, int nSize, long lType);
```

```
    HRESULT GetCrossbarIPinAtIndex(
              IAMCrossbar *pXbar,
              LONG PinIndex,
              BOOL IsInputPin,
              IPin ** ppPin);
    HRESULT GetCrossbarIndexFromIPin (
              IAMCrossbar * pXbar,
              LONG * PinIndex,
              BOOL IsInputPin,
              IPin * pPin);

public:  // 公共接口函数
    CCrossbar (IPin *pPin, HRESULT *phr);
    ~CCrossbar();
    // 得到输入端子的个数
HRESULT GetInputCount (LONG *pCount);
    // 得到指定索引的输入端子的物理类型
HRESULT GetInputType  (LONG Index, LONG * PhysicalType);
    // 得到指定索引的输入端子的名字
HRESULT GetInputName  (LONG Index, TCHAR * pName, LONG NameSize);
    // 通过索引设置输入端子
HRESULT SetInputIndex (LONG Index);
    // 得到当前输入端子的索引
    HRESULT GetInputIndex (LONG *Index);
};

#endif // __Crossbar_h__
```

AVCap程序的CAnalogInputFilters类中演示了CCrossbar类的使用方法。

```
//
// CAnalogInputFilters.h
//
class CCrossbar;
class CAnalogInputFilters : public CLiveInputFilters
{
protected:
    CCrossbar * mCrossbar;  // 定义一个CCrossbar对象指针
    // 其他类成员定义（省略）
    // ……
public:
    CAnalogInputFilters(CDXGraph * inGraph);
    virtual ~CAnalogInputFilters();
public:
    virtual BOOL Create(void);
    // 视频输入端子相关的操作函数
    virtual long GetVideoConnectorCount(void);
    virtual CString GetVideoConnectorName(long inIndex);
    virtual long GetVideoConnectorType(long inIndex);
    virtual void SetVideoConnector(long inPhysicalType);
    // 音频输入端子相关的操作函数
    virtual long GetAudioConnectorCount(void);
    virtual CString GetAudioConnectorName(long inIndex);
```

```
    virtual void SetAudioConnector(long inIndex);
    virtual void SetAudioMixLevel(double inLevel);
    // 其他接口函数声明（省略）
    // ……
};

//
// CAnalogInputFilters.cpp
//
// 采集设备输入部分的创建函数
BOOL CAnalogInputFilters::Create(void)
{
    // 创建采集设备Filter（省略）
    // ……
    // 成功创建采集设备Filter后进行一系列参数设置
    // 这里创建Crossbar的包装对象（生成输入端子选择的映射信息）
    CreateCrossBar(mVideoCapture->GetFilter());
    // 其他参数设置（省略）
    // ……
}
// 在采集设备 Filter上找到“Analog Video In”输入Pin,
// 然后以这个Pin为参数创建一个CCrossbar对象
BOOL CAnalogInputFilters::CreateCrossBar(IBaseFilter * inFilter)
{
    IEnumPins * pins = NULL;
    HRESULT hr = inFilter->EnumPins(&pins);
    if (FAILED(hr))
    {
        return FALSE;
    }

    IPin * pPin= NULL;
    IKsPropertySet * pKs = NULL;
    ULONG       count    = 0;
    BOOL        found    = FALSE;
    PIN_INFO    pinInfo;
    // 枚举Filter上所有的Pin
    while (!found && SUCCEEDED(pins->Next(1, &pPin, &count)) && count)
    {
        if (S_OK == pPin->QueryPinInfo(&pinInfo))
        {
            // 判断当前得到的Pin是不是输入Pin
            if (pinInfo.dir == PINDIR_INPUT)
            {
                // 查询当前Pin上的属性集
                if (pPin->QueryInterface(IID_IKsPropertySet,
                    (void **)&pKs) == S_OK)
                {
                    GUID  guid;
                    DWORD dw = 0;
                    // 得到当前Pin的类型
                    if(pKs->Get(AMPROPSETID_Pin, AMPROPERTY_PIN_CATEGORY,
```

```
                    0, 0,&guid, sizeof(GUID), &dw) == S_OK)
                {
                    // 判断是不是“Analog Video In” Pin
                    if (guid == PIN_CATEGORY_ANALOGVIDEOIN)
                    {
                        found = TRUE;
                    }
                }
                pKs->Release();
            }
            if (found)
            {
                // 如果找到了“Analog Video In” Pin,
                // 则创建一个CCrossbar对象
                SAFE_DELETE(mCrossbar);
                HRESULT hr = S_OK;
                mCrossbar  = new CCrossbar(pPin, &hr);
            }
        }
        pinInfo.pFilter->Release();
    }
    pPin->Release();
  }
  pins->Release();
  return found;
}
// 得到视频输入端子的个数
long CAnalogInputFilters::GetVideoConnectorCount(void)
{
    long count = 0;
    if (mCrossbar)
    {
        mCrossbar->GetInputCount(&count);
    }
    return count;
}
// 得到指定索引的视频输入端子的名字
CString CAnalogInputFilters::GetVideoConnectorName(long inIndex)
{
    char connectorName[265];
    strcpy(connectorName, "");
    if (mCrossbar)
    {
        mCrossbar->GetInputName(inIndex, connectorName, 265);
    }
    return connectorName;
}
// 得到指定索引的视频输入端子的物理类型
long CAnalogInputFilters::GetVideoConnectorType(long inIndex)
{
    long physicalType = 0;
    if (mCrossbar)
```

```
        {
            mCrossbar->GetInputType(inIndex, &physicalType);
        }
        return physicalType;
    }
    // 通过物理类型来设置视频输入端子
    void CAnalogInputFilters::SetVideoConnector(long inPhysicalType)
    {
        if (mCrossbar)
        {
            long count       = 0;
            long physicalType = 0;
            // 遍历所有视频输入端子进行物理类型的匹配
            mCrossbar->GetInputCount(&count);
            for (long i = 0; i < count; i++)
            {
                mCrossbar->GetInputType(i, &physicalType);
                if (physicalType == inPhysicalType)
                {
                    // 通过索引选择一个输入端子
                    mCrossbar->SetInputIndex(i);
                    break;
                }
            }
        }
    }
```

WDM设备可以通过上述Crossbar操作在程序中实现视频输入端子的选择。VFW设备就没有那么幸运了。为VFW设备选择输入端子，必须把VFW驱动程序定制的标准的设置对话框显示出来。VFW设备的设置对话框包括Video Source（设置图像源属性）、Video Format（设置图像输出格式）和Video Display（设置图像显示参数）。显示这些对话框的参考代码如下：

```
//
// CAnalogInputFilters.cpp
//
// 3种VFW设置对话框的类型定义
typedef enum {
    VfwCaptureDialog_Source  = 0x01,
    VfwCaptureDialog_Format  = 0x02,
    VfwCaptureDialog_Display = 0x04
} VfwCaptureDialogs;

// 判断指定类型的对话框是否可用
// 参数inDlgType的取值为上面那个枚举类型
BOOL CAnalogInputFilters::VFWHasDialog(int inDlgType)
{
    if (mVideoCapture && mVideoCapture->GetFilter())
    {
        // 从采集设备Filter上获取IAMVfwCaptureDialogs接口
        IAMVfwCaptureDialogs * pVfw = NULL;
```

```
        mVideoCapture->GetFilter()->QueryInterface(
            IID_IAMVfwCaptureDialogs,(void**)&pVfw);
        if (pVfw)
        {
            pVfw->Release();
            // 查询指定的对话框是否可用
            if (pVfw->HasDialog(inDlgType) == S_OK)
            {
                return TRUE;
            }
        }
    }
    return FALSE;
}
// 将指定类型的对话框显示出来
// 参数inDlgType的取值为上面那个枚举类型，hwnd为父窗口句柄
void CAnalogInputFilters::VFWShowDialog(int inDlgType, HWND hwnd)
{
    if (mGraph)
    {
        // 首先要将Filter Graph停止
        BOOL isStopped = mGraph->IsStopped();
        if (!isStopped)
        {
            mGraph->Stop();
        }

        // 从采集设备Filter上获取IAMVfwCaptureDialogs接口
        IAMVfwCaptureDialogs * pVfw = NULL;
        mVideoCapture->GetFilter()->QueryInterface(
            IID_IAMVfwCaptureDialogs,(void**)&pVfw);
        if (pVfw)
        {
            pVfw->Release();
            // 显示指定类型的设置对话框
            pVfw->ShowDialog(inDlgType, hwnd);
        }

        // 重新运行Filter Graph
        if (!isStopped)
        {
            mGraph->Run();
        }
    }
}
```

或者，干脆将采集设备Filter的属性页显示出来：

```
//
// CAnalogInputFilters.cpp
//
// 显示采集设备Filter的属性页
void CAnalogInputFilters::ShowVideoCapturePropertyPage(void)
```

```
{
    UDsUtils::ShowFilterPropertyPage(mVideoCapture->GetFilter());
}

//
// UDsUtils.cpp
//
// 将指定Filter的属性页显示出来
BOOL UDsUtils::ShowFilterPropertyPage(IBaseFilter * inFilter)
{
    if (!inFilter)
    {
        return FALSE;
    }
    // 从Filter上获得ISpecifyPropertyPages接口
    ISpecifyPropertyPages * pSpecify;
    HRESULT hr = inFilter->QueryInterface(IID_ISpecifyPropertyPages,
        (void **)&pSpecify);
    if (SUCCEEDED(hr))
    {
        CAUUID  caGUID;
        pSpecify->GetPages(&caGUID);
        pSpecify->Release();
        // 将属性页显示出来
        OleCreatePropertyFrame(
            ::GetActiveWindow(),        // 父窗口
            0,                          // x坐标
            0,                          // y坐标
            NULL,                       // 对话框标题
            1,                          // Filter个数
            (IUnknown **) &inFilter,    // Filter对象指针
            caGUID.cElems,              // 属性页个数
            caGUID.pElems,              // 属性页的CLSID
            0,
            0,
            NULL
            );
        CoTaskMemFree(caGUID.pElems); // 释放资源
        return TRUE;
    }
    return FALSE;
}
```

下面顺便再来看一下音频输入端子的选择。AVCap程序在成功创建音频采集设备后，会将所有的音频输入端子信息保存到一个列表中，并试图设置一个默认的输入端子Line In。但是有一个问题，不同声卡表示Line In端口的名字可能不一样，另外安装不同语言版本的声卡驱动程序后也会导致各输入端子名字的改变。似乎没有更好的办法去选择一个特定的音频输入端子，除了用一个一个相近的名字去猜测。

```
//
// CAudioCaptureFilter.cpp
```

```
//
// 设置默认的音频输入端子
void CAudioCaptureFilter::SetDefaultInputPin(void)
{
    IEnumPins * pinEnum = NULL;
    if (FAILED(mFilter->EnumPins(&pinEnum)))
    {
        return;
    }

    IPin * pin       = NULL;
    ULONG fetchCount = 0;
    BOOL  found      = FALSE;
    pinEnum->Reset();
    // 枚举音频采集设备Filter所有的Pin
    while (!found && SUCCEEDED(pinEnum->Next(1, &pin, &fetchCount)) &&
        fetchCount)
    {
        if (pin)
        {
            PIN_INFO  pinInfo;
            // 得到当前Pin的信息
            if (SUCCEEDED(pin->QueryPinInfo(&pinInfo)))
            {
                pinInfo.pFilter->Release();
                // 这是一个输入Pin?
                if (pinInfo.dir == PINDIR_INPUT)
                {
                    char szName[128];
                    ::WideCharToMultiByte(CP_ACP, 0, pinInfo.achName,
                        -1, szName, 128, NULL, NULL);

                    CString  pinName = szName;
                    // 猜测各种Line In端口的名字
                    if (pinName.CompareNoCase("Line In") == 0 ||
                        pinName.CompareNoCase("Line-In") == 0 ||
                        pinName.CompareNoCase("Line_In") == 0 ||
                        pinName.CompareNoCase("LineIn") == 0 ||
                        pinName.CompareNoCase("Line") == 0)
                    {
                        found = TRUE;
                    }
                    if (found)
                    {
                        // 如果找到了Line In Pin，则使用它
                        IAMAudioInputMixer * pMixer = GetMixer(pin);
                        pMixer->put_Enable(TRUE);
                        pMixer->put_MixLevel(1.0);
                    }
                }
            }
            pin->Release();
```

```
            }
        }
        pinEnum->Release();

        // 如果Line In Pin没有找到，则设置第一个输入端子
        if (!found)
        {
            SetConnector(0);
        }
    }
    // 获取音频采集设备Filter输入Pin上的IAMAudioInputMixer接口
    IAMAudioInputMixer * CAudioCaptureFilter::GetMixer(IPin * inPin)
    {
        IAMAudioInputMixer * pMixer = NULL;
        inPin->QueryInterface(IID_IAMAudioInputMixer, (void**) &pMixer);
        if (pMixer)
        {
            pMixer->Release();
        }
        return pMixer;
    }
    // 通过索引设置音频输入端子
    // mInputList是保存所有音频输入端子信息的一个列表，列表元素为CAudioInput类的对象
    // CAudioInput类有两个成员：CString  mInputName、 IPin *   mInputPin;
    void CAudioCaptureFilter::SetConnector(long inIndex)
    {
        if (inIndex >= 0 && inIndex < mInputList.GetCount())
        {
            // 找到指定索引的列表元素
            long index  = inIndex;
            POSITION pos = mInputList.GetHeadPosition();
            while (pos && index > 0)
            {
                mInputList.GetNext(pos);
                index--;
            }
            // 设置使用这个输入端子，即调用这个Pin的IAMAudioInputMixer::put_Enable
            if (pos)
            {
                CAudioInput inputItem = mInputList.GetNext(pos);
                IAMAudioInputMixer * pMixer = GetMixer(inputItem.mInputPin);
                pMixer->put_Enable(TRUE);
                pMixer->put_MixLevel(AMF_AUTOMATICGAIN);
            }
        }
    }
```

2.2.6 采集参数设置

首先来看一下视频输入信号的制式问题。视频信号的制式主要有3种：PAL、NTSC和SECAM。用户有责任将实际输入信号的制式正确地设置给采集设备，否则采集设备就不能

进行正确的解码，对比效果如图2.14所示（左边为制式不匹配情况下显示的图像，右边为制式匹配情况下显示的图像）。

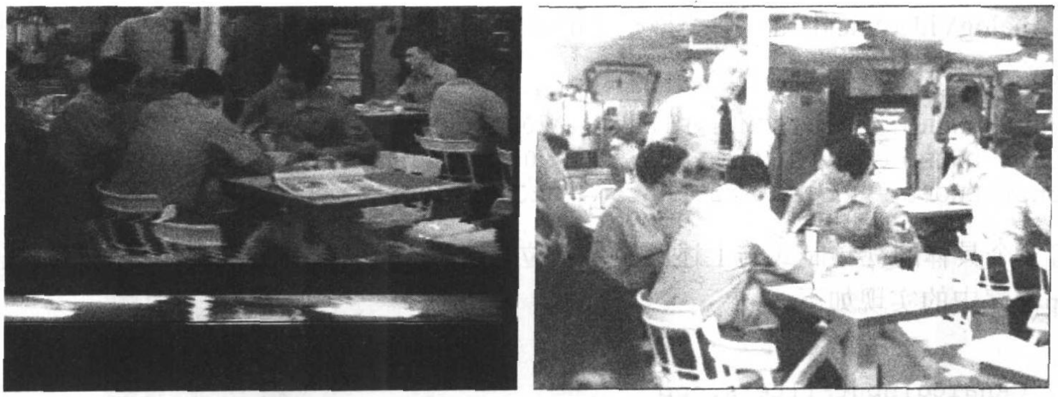


图 2.14 视频信号制式设置的正确与否对图像显示的影响

可以通过采集设备Filter上的IAMAnalogVideoDecoder接口的get_AvailableTVFormats函数，来获取采集设备支持的所有视频制式。该函数返回的是一个位或（Bitwise OR）组合的枚举类型，这个枚举类型定义如下：

```
typedef enum tagAnalogVideoStandard {
    AnalogVideo_None           = 0x00000000
    AnalogVideo_NTSC_M         = 0x00000001,
    AnalogVideo_NTSC_M_J       = 0x00000002,
    AnalogVideo_NTSC_433       = 0x00000004,
    AnalogVideo_PAL_B          = 0x00000010,
    AnalogVideo_PAL_D          = 0x00000020,
    AnalogVideo_PAL_H          = 0x00000080,
    AnalogVideo_PAL_I          = 0x00000100,
    AnalogVideo_PAL_M          = 0x00000200,
    AnalogVideo_PAL_N          = 0x00000400,
    AnalogVideo_PAL_60         = 0x00000800,
    AnalogVideo_SECAM_B        = 0x00001000,
    AnalogVideo_SECAM_D        = 0x00002000,
    AnalogVideo_SECAM_G        = 0x00004000,
    AnalogVideo_SECAM_H        = 0x00008000,
    AnalogVideo_SECAM_K        = 0x00010000,
    AnalogVideo_SECAM_K1       = 0x00020000,
    AnalogVideo_SECAM_L        = 0x00040000,
    AnalogVideo_SECAM_L1       = 0x00080000,
    AnalogVideo_PAL_N_COMBO    = 0x00100000
} AnalogVideoStandard;
```

上述枚举类型定义的是各种具体的NTSC、PAL、SECAM制式，但可能用户并不关心这些细节。于是，AVCap程序重新定义了一种枚举类型（只区分制式的大类）：

```
typedef enum
{
    SR_NTSC,
    SR_PAL,
```

```
        SR_SECAM,
        SR_UNKNOWN
    } Signal_Resolution;
```

然后从AnalogVideoStandard枚举类型定义的各种制式，得出NTSC、PAL、SECAM 3种制式的屏蔽字分别为：

```
    const DWORD    cNTSCMaskBits  = 0x0000000f;
    const DWORD    cPALMaskBits   = 0x00100ff0;
    const DWORD    cSECAMMaskBits = 0x000ff000;
```

只要将某个具体的视频制式与上述屏蔽字作位与操作，就可以得出该制式对应的制式大类。AVCap程序中的实现如下：

```
    //
    // CAnalogInputFilters.cpp
    //
    // 获取采集设备当前工作的视频制式
    Signal_Resolution CAnalogInputFilters::GetVideoResolution(void)
    {
        // 从采集设备Filter上获取IAMAnalogVideoDecoder接口
        IAMAnalogVideoDecoder * pVideoDecoder = GetAnalogDecoder();
        if (pVideoDecoder)
        {
            long format = 0;
            pVideoDecoder->get_TVFormat(&format);
            // 将具体的视频制式转换成自定义的（大类）制式
            if (cNTSCMaskBits & format)
            {
                return SR_NTSC;
            }
            else if (cPALMaskBits & format)
            {
                return SR_PAL;
            }
            else if (cSECAMMaskBits & format)
            {
                return SR_SECAM;
            }
        }
        return SR_UNKNOWN;
    }
    // 设置采集设备工作的视频制式
    void CAnalogInputFilters::SetVideoResolution(Signal_Resolution
        inResolution)
    {
        long  formatToSet = 0;
        // 获取采集设备支持的所有视频制式
        long  allFormats  = GetAllTVFormats();
        // 选择一种具体的视频制式
        if (inResolution == SR_NTSC)
        {
            if (AnalogVideo_NTSC_M & allFormats)
```

```
        {
            formatToSet = AnalogVideo_NTSC_M;
        }
        else if (AnalogVideo_NTSC_M_J & allFormats)
        {
            formatToSet = AnalogVideo_NTSC_M_J;
        }
        else
        {
            formatToSet = AnalogVideo_NTSC_433;
        }
    }
    else if (inResolution == SR_PAL)
    {
        if (AnalogVideo_PAL_D & allFormats)
        {
            formatToSet = AnalogVideo_PAL_D;
        }
        else if (AnalogVideo_PAL_B & allFormats)
        {
            formatToSet = AnalogVideo_PAL_B;
        }
        else if (AnalogVideo_PAL_H & allFormats)
        {
            formatToSet = AnalogVideo_PAL_H;
        }
        else if (AnalogVideo_PAL_I & allFormats)
        {
            formatToSet = AnalogVideo_PAL_I;
        }
        else if (AnalogVideo_PAL_M & allFormats)
        {
            formatToSet = AnalogVideo_PAL_M;
        }
        else if (AnalogVideo_PAL_N & allFormats)
        {
            formatToSet = AnalogVideo_PAL_N;
        }
        else if (AnalogVideo_PAL_60 & allFormats)
        {
            formatToSet = AnalogVideo_PAL_60;
        }
        else
        {
            formatToSet = AnalogVideo_PAL_N_COMBO;
        }
    }
    else if (inResolution == SR_SECAM)
    {
        if (AnalogVideo_SECAM_B & allFormats)
        {
            formatToSet = AnalogVideo_SECAM_B;
```

```
        }
        else if (AnalogVideo_SECAM_D & allFormats)
        {
            formatToSet = AnalogVideo_SECAM_D;
        }
        else if (AnalogVideo_SECAM_G & allFormats)
        {
            formatToSet = AnalogVideo_SECAM_G;
        }
        else if (AnalogVideo_SECAM_H & allFormats)
        {
            formatToSet = AnalogVideo_SECAM_H;
        }
        else if (AnalogVideo_SECAM_K & allFormats)
        {
            formatToSet = AnalogVideo_SECAM_K;
        }
        else if (AnalogVideo_SECAM_K1 & allFormats)
        {
            formatToSet = AnalogVideo_SECAM_K1;
        }
        else if (AnalogVideo_SECAM_L & allFormats)
        {
            formatToSet = AnalogVideo_SECAM_L;
        }
        else
        {
            formatToSet = AnalogVideo_SECAM_L1;
        }
    }
    if (formatToSet != 0)
    {
        // 将一种具体的视频制式设置给采集设备
        IAMAnalogVideoDecoder * pVideoDecoder = GetAnalogDecoder();
        if (pVideoDecoder)
        {
            pVideoDecoder->put_TVFormat(formatToSet);
        }
    }
}
// 得到采集设备支持的所有视频制式
long CAnalogInputFilters::GetAllTVFormats(void)
{
    long allFormats = 0;
    // 得到采集设备Filter上的IAMAnalogVideoDecoder接口
    IAMAnalogVideoDecoder * pVideoDecoder = GetAnalogDecoder();
    if (pVideoDecoder)
    {
        pVideoDecoder->get_AvailableTVFormats(&allFormats);
    }
    return allFormats;
}
```

```
// 获取采集设备Filter上的IAMAnalogVideoDecoder接口
IAMAnalogVideoDecoder * CAnalogInputFilters::GetAnalogDecoder(void)
{
    IAMAnalogVideoDecoder * pVideoDecoder = NULL;
    if (mVideoCapture && mVideoCapture->GetFilter())
    {
        // 从采集设备Filter上直接查询IAMAnalogVideoDecoder接口
        mVideoCapture->GetFilter()->QueryInterface(
            IID_IAMAnalogVideoDecoder, (void**)&pVideoDecoder);
        if (pVideoDecoder)
        {
            pVideoDecoder->Release();
            return pVideoDecoder;
        }
    }
    return NULL;
}
```

值得注意的是，通过图2.2的界面改变了采集设备的工作制式后，可能显示的视频图像仍然不正常，此时最好进行Filter Graph的重构。

```
//
// AVCapDlg.cpp
//
// 自定义的消息处理函数
bool CAVCapDlg::Respond(NotifyT inNotification, CMsgStation * inStation,
            long inParam1, long inParam2)
{
    switch (inNotification)
    {
    // 其他消息处理（省略）
    // ……

    case cVideoResolutionChanged:
        // 当视频制式改变时对Filter Graph进行重构
        if (mController)
        {
            // 销毁当前Filter Graph
            mController->Deactive();
            // 构建一个新的Filter Graph
            mController->Activate();
        }
        return true;
    }

    return CMsgReceiver::Respond(inNotification, inStation, inParam1,
        inParam2);
}
```

另外，采集设备输出图像的格式、尺寸、视频帧率等属性都是可以定制的。这些属性主要通过Capture（或Preview）输出Pin上的媒体类型来决定。AVCap程序中，首先为输出Pin选择一种媒体类型，然后改变这个媒体类型的格式细节，最后再设置回输出Pin上。具

体实现如下：

```
//
// CAnalogInputFilters.cpp
//
// 调整采集设备Filter的输出Pin上的媒体类型
void CAnalogInputFilters::AdjustOutput(void)
{
    HRESULT  hr        = S_OK;
    AM_MEDIA_TYPE * pmt = NULL;
    // 如果优先使用VMR，则选择一种YUV的媒体类型
#ifdef _Trying_VMR
    if (mLiveCapture->IsVMRPreferred())
    {
        pmt = SelectMediaTypeForVMR();
    }
#endif // _Trying_VMR
    // 如果不优先使用VMR，则选择一种RGB的媒体类型
    if (!pmt)
    {
        pmt = SelectMediaType();
    }
    // 调整视频帧率（PAL制为25帧/秒，NTSC制为29.97帧/秒）
    LONGLONG avgTimePerFrame = 400000;      // 25fps
    if (GetVideoResolution() == SR_NTSC)
    {
        avgTimePerFrame = 333667;      // 29.97fps
    }
    if (pmt)
    {
        if (pmt->formattype == FORMAT_VideoInfo)
        {
            // 获取欲调整的图像尺寸（由参数对象保存）
            long width, height;
            mLiveCapture->GetImageSize(&width, &height);

            VIDEOINFOHEADER *pvi = (VIDEOINFOHEADER *) pmt->pbFormat;
            pvi->AvgTimePerFrame      = avgTimePerFrame; // 调整帧率
            pvi->bmiHeader.biWidth     = width;   // 调整图像宽度
            pvi->bmiHeader.biHeight    = height;  // 调整图像高度
            pvi->bmiHeader.biSizeImage=width*height*pvi->
                bmiHeader.biBitCount/8;
            // 为Capture输出Pin设置新的媒体类型
            IAMStreamConfig * pCfg =
                        GetStreamConfig(&PIN_CATEGORY_CAPTURE);
            hr = pCfg->SetFormat(pmt);
            if (mHasPreviewPin)
            {
                // 同时为Preview输出Pin设置新的媒体类型
                pCfg = GetStreamConfig(&PIN_CATEGORY_PREVIEW);
                hr = pCfg->SetFormat(pmt);
            }
```

```
        }
        DeleteMediaType(pmt);
    }
}
```

在上述的设置过程中，明确地指定了要求采集设备输出图像的宽和高。但是，一般采集设备支持的输出图像尺寸是有一定范围的，设置一种超出范围的图像尺寸将会失败。有时候，可能只希望采集设备输出一种它支持的最大图像尺寸，具体实现如下：

```
//
// CAnalogInputFilters.cpp
//
// 调整采集设备Filter，以输出最大的图像尺寸
void CAnalogInputFilters::AdjustOutputToMax(void)
{
    VIDEO_STREAM_CONFIG_CAPS  pSCC;
    AM_MEDIA_TYPE *   pmt   = NULL;
    LONGLONG avgTimePerFrame = 400000;     // 25fps
    if (GetVideoResolution() == SR_NTSC)
    {
        avgTimePerFrame = 333667;      // 29.97fps
    }
    // 获取Capture输出Pin上支持的第一个媒体类型
    IAMStreamConfig * pCfg = GetStreamConfig(&PIN_CATEGORY_CAPTURE);
    pCfg->GetStreamCaps(0, &pmt, (BYTE *)&pSCC);
    // 获取输出图像尺寸调整的粒度
    int  granularityX   = pSCC.OutputGranularityX;
    int  granularityY   = pSCC.OutputGranularityY;
    // 获取采集设备支持的输出图像的最小尺寸和最大尺寸
    SIZE minOutputSize  = pSCC.MinOutputSize;
    SIZE maxOutputSize  = pSCC.MaxOutputSize;
    if (pmt && (pmt->formattype == FORMAT_VideoInfo) &&
       (granularityX || granularityY))
    {
        bool   bTurnOfx = false;
        int    nStepx    = 0;
        int    nStepy    = 0;
        int    nCycles  = 0;
        const int nCycleLen = 5;
        long currentFormat = 0;
        HRESULT hr = S_OK;

        VIDEOINFOHEADER * pvi = (VIDEOINFOHEADER *) pmt->pbFormat;
        pvi->AvgTimePerFrame  = avgTimePerFrame;

        do
        {
            // 从采集设备支持的最大尺寸逐步下调
            pvi->bmiHeader.biWidth  = maxOutputSize.cx - nStepx *
                granularityX;
            pvi->bmiHeader.biHeight = maxOutputSize.cy - nStepy *
                granularityY;
```

```
            // 如果当前的图像尺寸小于最小尺寸，则跳出循环
            if (pvi->bmiHeader.biWidth < minOutputSize.cx ||
                pvi->bmiHeader.biHeight < minOutputSize.cy)
            {
                break;
            }
            // 交替下调输出图像的宽和高
            nCycles++;
            if (nCycles > nCycleLen)
            {
                bTurnOfx = !bTurnOfx;
                nCycles  = 1;
            }
            if (bTurnOfx)
            {
                nStepx++;
            }
            else
            {
                nStepy++;
            }
            // 将新的媒体类型设置给Capture输出Pin并且检查返回值，
            // 如果成功就可以结束循环
            hr = pCfg->SetFormat(pmt);

        } while (hr != S_OK);
    }
    // 销毁使用过的媒体类型
    if (pmt)
    {
        DeleteMediaType(pmt);
        pmt = NULL;
    }
}
```

采集设备输出图像的显示参数（亮度、色调、饱和度、对比度等）也可以调整（通过采集设备Filter上的IAMVideoProcAmp接口）。AVCap程序并没有定制自己的图像参数设置界面，图像参数的调整是直接通过视频采集设备Filter的属性页（见图2.3）来完成的。其实，采集设备Filter的属性页也是使用了IAMVideoProcAmp接口。AVCap程序给出了如下的演示代码：

```
//
// CAnalogInputFilters.cpp
//
// 图像显示参数的类型定义
typedef enum tagVideoProcAmpProperty
{
   VideoProcAmp_Brightness,              // 亮度
   VideoProcAmp_Contrast,                // 对比度
   VideoProcAmp_Hue,                     // 色调
   VideoProcAmp_Saturation,              // 饱和度
```

```
    VideoProcAmp_Sharpness,                // 锐利度
    VideoProcAmp_Gamma,                    // Gamma校正
    VideoProcAmp_ColorEnable,              // 颜色使能
    VideoProcAmp_WhiteBalance,             // 白平衡
    VideoProcAmp_BacklightCompensation,    // 背景光补偿
    VideoProcAmp_Gain                      // 增益
} VideoProcAmpProperty;

// 得到指定的某个图像参数的取值范围
// 函数参数Property指定为上述枚举类型VideoProcAmpProperty中的一个值
BOOL CAnalogInputFilters::ImageAdjust_GetRange(long Property, long *pMin,
    long *pMax, long *pSteppingDelta, long *pDefault, long *pCapsFlags)
{
    if (mVideoCapture && mVideoCapture->GetFilter())
    {
        // 从采集设备Filter上获得IAMVideoProcAmp接口
        IAMVideoProcAmp * pAmp = NULL;
        mVideoCapture->GetFilter()->QueryInterface(IID_IAMVideoProcAmp,
            (void**)&pAmp);
        if (pAmp)
        {
            // 调用IAMVideoProcAmp::GetRange获得值域
            HRESULT hr = pAmp->GetRange(Property, pMin, pMax,
                pSteppingDelta, pDefault, pCapsFlags);
            pAmp->Release();
            return SUCCEEDED(hr);
        }
    }
    return FALSE;
}
// 设置指定的某个图像参数的值
BOOL CAnalogInputFilters::ImageAdjust_Set(long Property, long lValue, long
    Flags)
{
    if (mVideoCapture && mVideoCapture->GetFilter())
    {
        // 从采集设备Filter上获得IAMVideoProcAmp接口
        IAMVideoProcAmp * pAmp = NULL;
        mVideoCapture->GetFilter()->QueryInterface(IID_IAMVideoProcAmp,
            (void**)&pAmp);
        if (pAmp)
        {
            // 调用IAMVideoProcAmp::Set设置指定图像参数的值
            HRESULT hr = pAmp->Set(Property, lValue, Flags);
            pAmp->Release();
            return SUCCEEDED(hr);
        }
    }
    return FALSE;
}
// 获取指定的某个图像参数的值
BOOL CAnalogInputFilters::ImageAdjust_Get(long Property, long *lValue,
```

```
    long *Flags)
{
    if (mVideoCapture && mVideoCapture->GetFilter())
    {
        // 从采集设备Filter上获得IAMVideoProcAmp接口
        IAMVideoProcAmp * pAmp = NULL;
        mVideoCapture->GetFilter()->QueryInterface(IID_IAMVideoProcAmp,
            (void**)&pAmp);
        if (pAmp)
        {
            // 调用IAMVideoProcAmp::Get获得指定图像参数的值
            HRESULT hr = pAmp->Get(Property, lValue, Flags);
            pAmp->Release();
            return SUCCEEDED(hr);
        }
    }
    return FALSE;
}
```

提示：典型的情况下，DirectShow数据处理的整个流程就是：输入源 -> 中间件 -> 输出终端。在上述这3个步骤中，都有机会对图像的显示参数进行调整（假设Filter Graph中含有视频流）。采集设备上的调整实际上属于输入源这个步骤，除此之外，在另外两个步骤中也可以进行。比如使用VMR-9进行视频显示的时候，就可以利用该Filter上的IVMRMixerControl9接口进行图像参数的调整（GetProcAmpControlRange：得到指定图像参数的取值范围，SetProcAmpControl：设置图像参数值，GetProcAmpControl：得到当前的图像参数值）。当然，VMR-9的这个功能必须得到显卡的支持（传统的Video Renderer和VMR-7都没有提供类似的功能）。而如果在中间件这个步骤去调整图像参数的话，就必须自己开发一个Transform Filter来完成。通常这样的中间件要求实现的效率比较高，以满足实时性要求。（对于一般的媒体文件，如果在播放过程中也想对图像的显示作出一些调整，则要么开发/插入一个实现这种功能的中间件，要么指定使用VMR-9来进行视频显示。）

下面，顺便再来看一下音频采集参数的设置。当音频采集设备成功创建之后，可以调整采集缓存的大小，以减小声音的延时。默认情况下，音频采集设备Filter使用500ms大小的缓存：只有采集满了500ms的数据，Filter才会发送出一个Sample。对于一些特殊的应用，500ms的延时可能是不能容忍的。一般缓存大小设置成能够容纳80ms的数据就可以了，甚至可以更少（也不能太少，否则会影响音频采集的效率，严重时还会损害音质）。

```
//
// CAudioCaptureFilter.cpp
//
// 设置音频采集的缓存大小
void CAudioCaptureFilter::SetCaptureBufferSize(void)
{
    // 得到音频采集设备Filter的Capture输出Pin
    IPin * pCapturePin = GetPin(FALSE);
```

```
    if (pCapturePin)
    {
        DWORD  dwBytesPerSec = 0;
        DWORD  dwLatencyInMilliseconds = 40;  // 40ms的缓存大小
        AM_MEDIA_TYPE * pmt = {0};
        IAMStreamConfig * pCfg = NULL;
        HRESULT hr = pCapturePin->QueryInterface(IID_IAMStreamConfig,
            (void **)&pCfg);
        if (SUCCEEDED(hr))
        {
            // 得到音频采集设备Filter的输出Pin上的媒体类型
            hr = pCfg->GetFormat(&pmt);
            if (SUCCEEDED(hr))
            {
                // 获取采集输出每秒钟的数据量
                WAVEFORMATEX *pWF = (WAVEFORMATEX *) pmt->pbFormat;
                dwBytesPerSec = pWF->nAvgBytesPerSec;
                DeleteMediaType(pmt);
            }
            pCfg->Release();
        }
        if (dwBytesPerSec)
        {
            IAMBufferNegotiation * pNeg = NULL;
            hr = pCapturePin->QueryInterface(IID_IAMBufferNegotiation,
                (void **)&pNeg);
            if (SUCCEEDED(hr))
            {
                ALLOCATOR_PROPERTIES AllocProp;
                AllocProp.cbAlign  = -1;
                // 计算新的缓存大小
                AllocProp.cbBuffer= dwBytesPerSec *
                    dwLatencyInMilliseconds / 1000;
                AllocProp.cbPrefix = -1;
                AllocProp.cBuffers = -1;
                // 向音频采集设备Filter建议使用新尺寸的缓存
                hr = pNeg->SuggestAllocatorProperties(&AllocProp);
                pNeg->Release();
            }
        }
    }
}
```

当用户为音频采集选择了一个输入端子之后，通常还可以给当前这个端子设置录音的音量大小。设置的方法就是获得该输入端子（对应的输入Pin）上的IAMAudioInputMixer接口，然后调用它的接口函数put_MixLevel，实现如下：

```
//
// CAudioCaptureFilter.cpp
//
// 给所有的输入端子设置音量
// 参数inLevel的取值范围0.0-1.0
```

```
void CAudioCaptureFilter::SetMixLevel(double inLevel)
{
    POSITION pos = mInputList.GetHeadPosition();
    while (pos)
    {
        CAudioInput inputItem = mInputList.GetNext(pos);
        // 获得输入Pin上的IAMAudioInputMixer接口
        IAMAudioInputMixer * pMixer = GetMixer(inputItem.mInputPin);
        pMixer->put_MixLevel(inLevel);
    }
}
```

2.2.7 支持数码摄像机

AVCap程序除了支持AGP、PCI、USB等接口的采集设备外，还支持数码摄像机（DV）的输入。数码摄像机通过1394接口与计算机相连，并且可以作为视频采集设备而被枚举到，它的友好名字为Microsoft DV Camera and VCR。

数码摄像机一般可以工作在两种模式：摄像头模式和磁带模式。前者相当于一个摄像头的功能，实时摄取音视频内容，并可以输出DV格式的数据（保存到磁带上或通过1394线传送给计算机）；后者播放磁带上内容，也是输出DV格式的数据（在摄像机的显示屏上显示或通过1394线传送给计算机）。数码摄像机预览用的Filter Graph有2种（如图2.15和图2.16所示）。

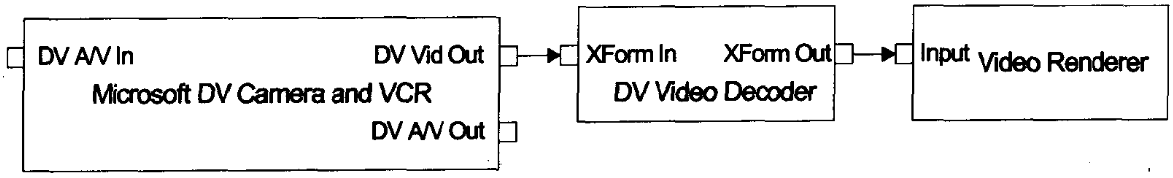


图 2.15　仅播放视频的数码摄像机 Filter Graph

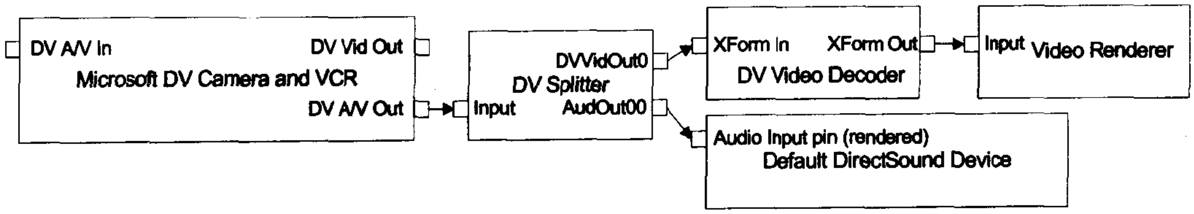


图 2.16　同时播放音视频的数码摄像机 Filter Graph

AVCap程序使用了一个包装类CDVCaptureFilter来负责数码摄像机Filter的创建，以及与摄像机直接相关的一些操作（通过摄像机Filter上的IAMExtDevice、IAMExtTransport、IAMTimecodeReader等接口完成）。

提示：CDVCaptureFilter类封装了对数码摄像机的一系列控制函数，然而这些函数在AVCap程序中并没有用到。读者有兴趣的话，可以使用控制函数来实现一个数码摄像机的操作界面，很有意思！

另外，AVCap程序使用了一个CDVInputFilters类来辅助数码摄像机Filter Graph的构建。CDVInputFilters类的定义和实现如下：

```
//
// CDVInputFilters.h
//

#ifndef __H_CDVInputFilters__
#define __H_CDVInputFilters__

#include "CLiveInputFilters.h"

class CDXFilter;
class CDVCaptureFilter;
class CDVInputFilters : public CLiveInputFilters
{
private:
    CDVCaptureFilter *  mDVCapture;          // 数码摄像机
    CDXFilter *     mDVSplitter;             // DV音视频分离器
    CDXFilter *     mDVVideoDecoder;         // DV视频解码器
public:
    CDVInputFilters(CDXGraph * inGraph);
    virtual ~CDVInputFilters();
public:
    virtual BOOL Create(void);
    // 是场图像吗?
    virtual BOOL IsFieldPicture(void);
    // 得到视频的采集、预览输出Pin（主要用于连接）
    virtual IPin * GetVideoOutPin(void);
    virtual IPin * GetPreviewPin(void);
    // 得到音频的输出Pin（主要用于连接）
    virtual IPin * GetAudioOutPin(void);
    // 显示数码摄像机Filter的属性页
    virtual void ShowVideoCapturePropertyPage(void);
    // 判断当前使用的数码摄像机是否脱离系统（支持热插拔时使用）
    virtual BOOL IsActiveDeviceLost(IUnknown * pLost);
protected:
    virtual void Release(void);
    // 调整输出图像的尺寸
    void AdjustOutput(void);
    // 检查是否有必要插入一个Smart Tee Filter
    void CheckSmartTee(void);
};

#endif // __H_CDVInputFilters__

//
// CDVInputFilters.cpp
//

#include "stdafx.h"
#include "CDVInputFilters.h"
```

```
#include "CDXFilter.h"
#include "CDVCaptureFilter.h"
#include "CLiveCapture.h"
#include "CSmartTeeFilter.h"
#include "UDsUtils.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//////////////////////////////////////////////////////////////////////
CDVInputFilters::CDVInputFilters(CDXGraph * inGraph) :
CLiveInputFilters(inGraph)
{
    mDVCapture  = NULL;
    mDVSplitter = NULL;
    mDVVideoDecoder = NULL;
}
CDVInputFilters::~CDVInputFilters()
{
    // 销毁所有创建的Filter
    Release();
}
// 创建数码摄像机输入部分的Filter
BOOL CDVInputFilters::Create(void)
{
    ASSERT(mGraph);
    ASSERT(mLiveCapture);

    HRESULT hr = S_OK;
    // 创建数码摄像机Filter包装对象
    mDVCapture  = new CDVCaptureFilter(mGraph->GetGraph(),
        mLiveCapture->GetVideoDevice());
    // 创建Smart Tee Filter包装对象
    mSmartTee  = new CSmartTeeFilter(mGraph->GetGraph());
    // 创建DV Splitter Filter包装对象
    mDVSplitter = new CDXFilter(mGraph->GetGraph(), CLSID_DVSplitter,
        "DV Splitter");
    // 创建DV Video Decoder Filter包装对象
    mDVVideoDecoder = new CDXFilter(mGraph->GetGraph(),
        CLSID_DVVideoCodec,"DV Video Decoder");
    // 首先调用父类的Create函数
    BOOL pass = CLiveInputFilters::Create();
    if (pass)
    {
        // 真正创建数码摄像机Filter，并加入到Filter Graph
        pass = mDVCapture->CreateFilter();
    }
    if (pass)
```

```
    {
        // 真正创建DV Spiltter和DV Video Decoder Filter
        pass = mDVSplitter->CreateFilter();
        pass = mDVVideoDecoder->CreateFilter();
    }
    if (pass)
    {
        // 将数码摄像机与DV Splitter相连
        IPin * pOut = mDVCapture->GetAVOutputPin();
        IPin * pIn  = mDVSplitter->GetPin(TRUE);
        pass = mGraph->ConnectFilters(pOut, pIn);
        // 将DV Splitter与DV Video Decoder相连
        pOut = mDVSplitter->GetPin(FALSE, "DVVidOut0");
        pIn  = mDVVideoDecoder->GetPin(TRUE);
        pass = mGraph->ConnectFilters(pOut, pIn);
    }
    if (pass)
    {
        // 调整输出图像的尺寸
        AdjustOutput();
        // 必要时插入一个Smart Tee Filter
        CheckSmartTee();
    }
    return pass;
}
// 销毁使用过的各个Filter
// #define SAFE_DELETE(p)  { if(p) { delete (p);  (p)=NULL; } }
void CDVInputFilters::Release(void)
{
    SAFE_DELETE(mSmartTee);
    SAFE_DELETE(mDVSplitter);
    SAFE_DELETE(mDVVideoDecoder);
    SAFE_DELETE(mDVCapture);
}
// 当DV图像尺寸为720 x 480或720 x 576时，一幅图像有两场组成
BOOL CDVInputFilters::IsFieldPicture(void)
{
    return TRUE;
}
// 得到视频输出Pin (用于采集)
IPin * CDVInputFilters::GetVideoOutPin(void)
{
    // 如果连接了Smart Tee，则返回Smart Tee的Capture输出Pin
    if (mSmartTee && mSmartTee->GetFilter())
    {
        return mSmartTee->GetCapturePin();
    }
    // 如果没有连接Smart Tee，则返回DV Video Decoder的输出Pin
    if (mDVVideoDecoder->GetFilter())
    {
        return mDVVideoDecoder->GetPin(FALSE);
    }
```

```
        return NULL;
    }
    // 得到视频预览输出Pin
    IPin * CDVInputFilters::GetPreviewPin(void)
    {
        // 如果连接了Smart Tee，则返回Smart Tee的Preview输出Pin
        if (mSmartTee && mSmartTee->GetFilter())
        {
            return mSmartTee->GetPreviewPin();
        }
        // 如果没有连接Smart Tee，则返回DV Video Decoder的输出Pin
        if (mDVVideoDecoder->GetFilter())
        {
            return mDVVideoDecoder->GetPin(FALSE);
        }
        return NULL;
    }
    // 得到音频输出Pin
    IPin * CDVInputFilters::GetAudioOutPin(void)
    {
        // 返回DV Splitter的音频输出Pin
        if (mDVSplitter->GetFilter())
        {
            return mDVSplitter->GetPin(FALSE, "AudOut00");
        }
        return NULL;
    }
    // 显示数码摄像机Filter的属性页
    void CDVInputFilters::ShowVideoCapturePropertyPage(void)
    {
        UDsUtils::ShowFilterPropertyPage(mDVCapture->GetFilter());
    }
    // 判断是不是当前的设备丢失（支持热插拔时使用）
    BOOL CDVInputFilters::IsActiveDeviceLost(IUnknown * pLost)
    {
        if (mDVCapture && mDVCapture->GetFilter())
        {
            IUnknown * pDevice = NULL;
            // 从摄像机Filter上获得IUnknown接口指针
            mDVCapture->GetFilter()->QueryInterface(IID_IUnknown,
                    (void**)&pDevice);
            pDevice->Release();
            // 判断两个IUnknown对象指针指向的是不是同一个对象
            if (::IsEqualObject(pDevice, pLost))
            {
                return TRUE;
            }
        }
        return FALSE;
    }
    // DV Video Decoder仅支持4种不同的输出图像尺寸（具体尺寸根据信号制式而定）
    // DVRESOLUTION_FULL（完整大小），DVRESOLUTION_HALF（宽高都缩小一半），
```

```
// DVRESOLUTION_QUARTER（宽高都缩小为1/4），DVRESOLUTION_DC（宽88，高缩小为1/8）
void CDVInputFilters::AdjustOutput(void)
{
    _DVRESOLUTION dvResolution;
    // 取得欲设置的输出图像尺寸（由参数对象保存）
    long width = 0, height = 0;
    mLiveCapture->GetImageSize(&width, &height);
    // 选择一种最靠近目标图像尺寸的参数
    if (width >= 720)
    {
        dvResolution = DVRESOLUTION_FULL;
    }
    else if (width < 720 && width >= 360)
    {
        dvResolution = DVRESOLUTION_HALF;
    }
    else if (width < 360 && width >= 180)
    {
        dvResolution = DVRESOLUTION_QUARTER;
    }
    else
    {
        dvResolution = DVRESOLUTION_DC;
    }
    // 通过IIPDVDec接口设置输出图像的尺寸
    IBaseFilter * pFilter = mDVVideoDecoder->GetFilter();
    if (pFilter)
    {
        IIPDVDec * pIPDVDec = NULL;
        // 在DV Video Decoder上获得IIPDVDec接口
        pFilter->QueryInterface(IID_IIPDVDec, (void**)&pIPDVDec);
        if (pIPDVDec)
        {
            pIPDVDec->Release();
            pIPDVDec->put_IPDisplay(dvResolution);
        }
    }
}
// 如果是采集模式，则DV Video Decoder后面接一个Smart Tee Filter，以便在采集的同
// 时进行视频图像的预览
void CDVInputFilters::CheckSmartTee(void)
{
    if (mLiveCapture->GetWorkMode() == MD_Capture)
    {
        // 创建Smart Tee Filter并将其加入到Filter Graph
        BOOL pass = mSmartTee->CreateFilter();
        if (pass && mDVVideoDecoder->GetFilter())
        {
            // 在DV Video Decoder后面接上Smart Tee
            IPin * pOut = mDVVideoDecoder->GetPin(FALSE);
            IPin * pIn  = mSmartTee->GetInputPin();
            pass = mGraph->ConnectFilters(pOut, pIn);
```

```
        }
    }
}
```

提示：对数码摄像机输出的视频当然也可以使用VMR来显示，不过AVCap程序中并没有实现这个功能，有兴趣的读者可以把它当作练习来完成。

2.2.8 支持热插拔

一些采用热插拔接口（如USB、1394等）的采集设备随时都有可能脱离或接入系统，在程序中如何获取这些事件，进而做出相应的处理呢？这就是本节所要介绍的内容。

当一个正在被使用的采集设备脱离系统时，Filter Graph Manager会向应用程序发出一个EC_DEVICE_LOST事件；而当这个设备随后接入系统时，Filter Graph Manager会再次向应用程序发出一个EC_DEVICE_LOST事件。两次事件通过事件携带的第二个参数来区别：脱离时为0，接入时为1。实际上，代表采集设备的Filter在经历过脱离、接入的反复过程之后，内部状态不再保持，此时必须对Filter Graph进行重构。

另外一种情况，一个全新的采集设备在应用程序启动之后接入系统，此时就不能依赖Filter Graph Manager来处理热插拔事件了，而要依赖窗口消息WM_DEVICECHANGE。AVCap程序设计了一个窗口类CDeviceObserver，专门来处理这个WM_DEVICECHANGE消息。

```
//
// DeviceNotifyDefs.h
//

#ifndef __H_DeviceNotifyDefs__
#define __H_DeviceNotifyDefs__

#if (WINVER < 0x0500)
#define DBT_DEVTYP_DEVICEINTERFACE      0x00000005
#define DBT_DEVTYP_HANDLE               0x00000006
#define DEVICE_NOTIFY_WINDOW_HANDLE     0x00000000
typedef  PVOID          HDEVNOTIFY;
#endif

extern "C"
{
    // 接收WM_DEVICECHANGE消息需要注册
    // 注册和反注册函数的原型
    typedef BOOL (WINAPI *PUnregisterDeviceNotification)(IN HDEVNOTIFY
            Handle);
    typedef HDEVNOTIFY (WINAPI *PRegisterDeviceNotification)(IN HANDLE
            hRecipient,IN LPVOID NotificationFilter, IN DWORD Flags);
}

#if (WINVER < 0x0500)
typedef struct _DEV_BROADCAST_DEVICEINTERFACE {
    DWORD       dbcc_size;
```

```
    DWORD       dbcc_devicetype;
    DWORD       dbcc_reserved;
    GUID        dbcc_classguid;
    char        dbcc_name[1];
} DEV_BROADCAST_DEVICEINTERFACE, *PDEV_BROADCAST_DEVICEINTERFACE;
#endif

#endif // __H_DeviceNotifyDefs__

//
// CDeviceObserver.h
//

#ifndef __H_CDeviceObserver__
#define __H_CDeviceObserver__

#include "CMsgStation.h"
#include "DeviceNotifyDefs.h"

class CDeviceObserver : public CFrameWnd, public CMsgStation
{
protected:
    HDEVNOTIFY          mNotifyHandle;
    // 注册函数和反注册函数的入口地址
    PRegisterDeviceNotification mRegisterDeviceNotification;
    PUnregisterDeviceNotification  mUnregisterDeviceNotification;
    UINT      mEventType;     // 事件类型
    LPBYTE    mEventData;     // 事件参数
    CCritSec mStateLock;      // 同步对象
public:
    CDeviceObserver();
    virtual ~CDeviceObserver();
    void Register(void);     // 注册
    void Unregister(void);  // 反注册
    DECLARE_DYNCREATE(CDeviceObserver)
protected:
// Attributes
public:
// Operations
public:
// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CDeviceObserver)
    //}}AFX_VIRTUAL
// Implementation
protected:
    // Generated message map functions
    //{{AFX_MSG(CDeviceObserver)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnClose();
    // WM_DEVICECHANGE消息的响应函数
    afx_msg BOOL OnDeviceChange(UINT nEventType, DWORD dwData);
```

```
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

#endif // __H_CDeviceObserver__

//
// CDeviceObserver.cpp
//

#include "stdafx.h"
#include "Dbt.h"
#include <streams.h>
#include "CDeviceObserver.h"
#include "GlobalDefs.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CDeviceObserver
IMPLEMENT_DYNCREATE(CDeviceObserver, CFrameWnd)
CDeviceObserver::CDeviceObserver()
{
    mEventType = 0;
    mEventData = NULL;
}
CDeviceObserver::~CDeviceObserver()
{
    // 释放用于保存事件参数的内存资源
    if (mEventData)
    {
        delete[] mEventData;
        mEventData = NULL;
    }
}
// 将本窗口注册为接收WM_DEVICECHANGE消息
void CDeviceObserver::Register(void)
{
    // 初始化DEV_BROADCAST_DEVICEINTERFACE数据结构
    DEV_BROADCAST_DEVICEINTERFACE filterData;
    ZeroMemory(&filterData, sizeof(DEV_BROADCAST_DEVICEINTERFACE));
    filterData.dbcc_size = sizeof(DEV_BROADCAST_DEVICEINTERFACE);
    filterData.dbcc_devicetype = DBT_DEVTYP_DEVICEINTERFACE;
    filterData.dbcc_classguid = AM_KSCATEGORY_CAPTURE;

    mUnregisterDeviceNotification = NULL;
    mRegisterDeviceNotification   = NULL;
    // 从user32.dll中获取注册函数和反注册函数的入口地址
```

```
    {
        HMODULE hmodUser = GetModuleHandle(TEXT("user32.dll"));
        ASSERT(hmodUser);
        // 获得反注册函数
        mUnregisterDeviceNotification = (PUnregisterDeviceNotification)
            GetProcAddress(hmodUser, "UnregisterDeviceNotification");
        // 注册函数有两种名字:
        // RegisterDeviceNotificationW和RegisterDeviceNotificationA
        mRegisterDeviceNotification = (PRegisterDeviceNotification)
                        GetProcAddress(hmodUser,
#ifdef UNICODE
                        "RegisterDeviceNotificationW"
#else
                        "RegisterDeviceNotificationA"
#endif
                        );
        ASSERT(mRegisterDeviceNotification &&
              mUnregisterDeviceNotification ||
              !mRegisterDeviceNotification
              && !mUnregisterDeviceNotification);
    }

    mNotifyHandle = NULL;
    if (mRegisterDeviceNotification)
    {
        // 注册本窗口(对设备改变感兴趣)
        mNotifyHandle = mRegisterDeviceNotification(this->GetSafeHwnd(),
            &filterData, DEVICE_NOTIFY_WINDOW_HANDLE);
        ASSERT(mNotifyHandle != NULL);
    }
}
// 将本窗口注销接收WM_DEVICECHANGE消息
void CDeviceObserver::Unregister(void)
{
    if (mNotifyHandle != NULL)
    {
        ASSERT(mUnregisterDeviceNotification);
        mUnregisterDeviceNotification(mNotifyHandle);
        mNotifyHandle = NULL;
    }
}

// 窗口消息映射
BEGIN_MESSAGE_MAP(CDeviceObserver, CFrameWnd)
    //{{AFX_MSG_MAP(CDeviceObserver)
    ON_WM_CREATE()
    ON_WM_CLOSE()
    ON_WM_DEVICECHANGE()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

///////////////////////////////////////////////////////////////////////////
```

```
// CDeviceObserver message handlers
int CDeviceObserver::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    int createResult = CFrameWnd::OnCreate(lpCreateStruct);
    if (createResult != -1)
    {
        // 当窗口成功创建之后，注册为接收WM_DEVICECHANGE消息
        Register();
    }
    return createResult;
}
// 本窗口关闭的消息响应函数
void CDeviceObserver::OnClose()
{
    // 注销接收WM_DEVICECHANGE消息
    Unregister();
    CFrameWnd::OnClose();
}
// WM_DEVICECHANGE消息的响应函数
BOOL CDeviceObserver::OnDeviceChange(UINT nEventType, DWORD dwData)
{
    // 我们只对设备接入和脱离事件感兴趣
    if (DBT_DEVICEARRIVAL != nEventType && DBT_DEVICEREMOVECOMPLETE !=
        nEventType)
        return CWnd::OnDeviceChange (nEventType, dwData);

    PDEV_BROADCAST_HDR pdbh = (PDEV_BROADCAST_HDR) dwData;
    if (pdbh->dbch_devicetype != DBT_DEVTYP_DEVICEINTERFACE)
    {
        return CWnd::OnDeviceChange (nEventType, dwData);
    }
    PDEV_BROADCAST_DEVICEINTERFACE pdbi=(PDEV_BROADCAST_DEVICEINTERFACE)
        dwData;
    // 判断是不是采集设备
    if (pdbi->dbcc_classguid != AM_KSCATEGORY_CAPTURE)
    {
        return CWnd::OnDeviceChange (nEventType, dwData);
    }
    // 保存事件及事件参数
    mEventType = nEventType;
    if (mEventData)
    {
        delete[] mEventData;
        mEventData = NULL;
    }
    mEventData = new BYTE[pdbh->dbch_size];
    memcpy(mEventData, pdbh, pdbh->dbch_size);
    pdbi = (PDEV_BROADCAST_DEVICEINTERFACE) mEventData;

    if (nEventType == DBT_DEVICEARRIVAL)
    {
        // 如果是设备接入事件，则发送msg_PnPDeviceAdded消息让上层监听者处理
```

```
        // msg_PnPDeviceAdded为自定义消息，将设备的显示名字作为参数一起发送
        Broadcast(msg_PnPDeviceAdded, (void *)pdbi->dbcc_name, NULL);
    }
    else if (nEventType == DBT_DEVICEREMOVECOMPLETE)
    {
        // 如果是设备脱离事件，则发送msg_PnPDeviceRemoved消息让上层监听者处理
        // msg_PnPDeviceRemoved为自定义消息，将设备的显示名字作为参数一起发送
        Broadcast(msg_PnPDeviceRemoved, (void *)pdbi->dbcc_name, NULL);
    }
    return CWnd::OnDeviceChange (nEventType, dwData);
}
```

怎么来使用这个CDeviceObserver类呢？为简单起见，AVCap程序目前只对视频采集设备进行热插拔的支持。方法是，在用于维护视频采集设备信息列表的CVideoDevices类中创建一个CDeviceObserver类的实例；因为CDeviceObserver类是消息发送者（CDeviceObserver类从CMsgStation类继承），因此定义CVideoDevices类为消息接收者（CVideoDevices类从CMsgReceiver类继承），以便处理msg_PnPDeviceAdded消息和msg_PnPDeviceRemoved消息。另外，将CVideoDevices类同时从CMsgStation类继承，以便向主对话框类CAVCapDlg发送消息，通知它刷新界面上显示的设备列表，必要时进行Filter Graph的重构。

提示：进一步了解CMsgStation类和CMsgReceiver类之间的消息传递机制请参见2.4.1节。

```
//
// CVideoDevices.h
//

#ifndef __H_CVideoDevices__
#define __H_CVideoDevices__

#include "CDSDevice.h"
#include "CMsgStation.h"
#include "CMsgReceiver.h"

class CDeviceObserver;
class CVideoDevices : public CMsgStation, public CMsgReceiver
{
public:
    DEVICE_LIST        mList;        // 保存系统中所有视频采集设备信息的列表
private:
    static CVideoDevices * sDevices;    // 单件对象
protected:
    CDeviceObserver *   mWindow;        // 热插拔监测窗口
    CVideoDevices();
    ~CVideoDevices();
public:
    static CVideoDevices * Instance(void);        // 单件对象全局访问点
    static void Release(void);                    // 单件对象销毁函数
    // 枚举系统中所有的视频采集设备
    void Populate(void);
```

```
    // 根据设备的友好名字，从列表中找到这个设备
    BOOL FindDevice(const char * inDeviceName, CDSDevice& outDevice);
    // 从列表中增加/删除某设备
    BOOL AddDevice(const char * inDeviceName);
    void RemoveDevice(const char * inDeviceName);
    // 自定义消息响应函数
    virtual bool ReceiveMessage(MessageT inMessage, void * ioParam,
        void * ioParam2);
};

#endif // __H_CVideoDevices__

//
// CVideoDevices.cpp
//

#include "stdafx.h"
#include <streams.h>
#include "CVideoDevices.h"
#include "UDsUtils.h"
#include "CDeviceObserver.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

CVideoDevices * CVideoDevices::sDevices = NULL;  // 初始化
//////////////////////////////////////////////////////////////////////////
CVideoDevices::CVideoDevices()
{
    // 创建一个热插拔监测窗口
    mWindow = new CDeviceObserver();
    // CVideoDevices实例是热插拔监测窗口的消息接收者
    mWindow->AddMsgReceiver(this);
    mWindow->Create(0, "Devices", WS_POPUPWINDOW, CRect(0, 0, 2, 2), 0, 0);
    // 枚举系统中所有的视频采集设备
    Populate();
}
CVideoDevices::~CVideoDevices()
{
    // 销毁热插拔监测窗口
    mWindow->DestroyWindow();
    mList.RemoveAll();
}
// 单件对象全局访问点
CVideoDevices * CVideoDevices::Instance(void)
{
    if (sDevices == NULL)
    {
        sDevices = new CVideoDevices();
```

```
    }
    return sDevices;
}
// 单件对象销毁函数
void CVideoDevices::Release(void)
{
    if (sDevices)
    {
        delete sDevices;
        sDevices = NULL;
    }
}
// 枚举系统中所有的视频采集设备
void CVideoDevices::Populate(void)
{
    mList.RemoveAll();
    UDsUtils::QueryVideoCaptureDevices(mList);
}
// 根据设备的友好名字，从列表中找到这个设备
BOOL CVideoDevices::FindDevice(const char * inDeviceName, CDSDevice&
    outDevice)
{
    POSITION pos = mList.GetHeadPosition();
    while (pos)
    {
        CDSDevice device = mList.GetNext(pos);
        // 设备的友好名字匹配?
        if (device.GetDeviceFriendlyName().Compare(inDeviceName) == 0)
        {
            outDevice = device;
            return TRUE;
        }
    }
    return FALSE;
}
// 给出一个接入设备的显示名字，将这个设备信息追加到列表中去
BOOL CVideoDevices::AddDevice(const char * inDeviceName)
{
    char friendlyName[1024];
    // 根据设备的显示名字创建设备Filter，同时得到对应的友好名字
    IBaseFilter * pFilter = UDsUtils::CreateDevice(TRUE,
        inDeviceName, friendlyName);
    if (pFilter)
    {
        CDSDevice device;
        // 保存该设备的相关信息：显示名字、友好名字、设备类型等
        device.SetDeviceDisplayName(inDeviceName);
        device.SetDeviceFriendlyName(friendlyName);
        device.SetDevideType(UDsUtils::DecideDeviceType(pFilter));
        mList.AddTail(device);
        pFilter->Release();
        return TRUE;
```

```
    }
    return FALSE;
}
// 处理设备的脱离消息：将该设备从列表中删除
// inDeviceName: 脱离设备的显示名字
void CVideoDevices::RemoveDevice(const char * inDeviceName)
{
    POSITION pos = mList.GetHeadPosition();
    POSITION previousPos;
    while (pos)
    {
        previousPos = pos; // 暂存
        CDSDevice device = mList.GetNext(pos);
        // 设备的显示名字匹配
        // 注意使用UDsUtils::IsSameDevice函数，而不要直接比较字符串
        if (UDsUtils::IsSameDevice(device.GetDeviceDisplayName(),
            inDeviceName))
        {
            mList.RemoveAt(previousPos);
            break;
        }
    }
}
// 处理从热插拔监测窗口发送出来的消息：msg_PnPDeviceAdded和msg_PnPDeviceRemoved
bool CVideoDevices::ReceiveMessage(MessageT inMessage, void * ioParam, void
    * ioParam2)
{
    BOOL pass = TRUE;
    switch (inMessage)
    {
    case msg_PnPDeviceAdded:   // 设备接入事件
        // 将新设备加入到列表中保存
        pass = AddDevice((const char *)ioParam);
        break;

    case msg_PnPDeviceRemoved: // 设备脱离事件
        Sleep(100);
        // 将该设备从列表中删除
        RemoveDevice((const char *) ioParam);
        break;
    }

    if (pass)
    {
        // 将消息继续传送给上层监听者（主界面），以便作出相应的处理
        return Broadcast(inMessage, ioParam, ioParam2);
    }
    return true;
}

//
// AVCapDlg.cpp
```

```
//
// 主界面上自定义消息的处理函数
bool CAVCapDlg::ReceiveMessage(MessageT inMessage, void * ioParam, void *
    ioParam2)
{
    switch (inMessage)
    {
    case msg_PnPDeviceAdded:        // 设备接入事件
        FillVideoDevices();         // 刷新视频设备列表
        return true;

    case msg_PnPDeviceRemoved:      // 设备接入事件
        FillVideoDevices();         // 刷新视频设备列表
        return true;
    }
    return CMsgReceiver::ReceiveMessage(inMessage, ioParam, ioParam2);
}
```

对于热插拔的支持，通常处理WM_DEVICECHANGE消息就已经足够了。尽管如此，AVCap程序仍然演示了通过EC_DEVICE_LOST事件来支持的处理方法。（这是一种完全DirectShow的方法！）

```
//
// CDXGraph.cpp
//
// 在应用程序级别上处理Filter Graph Manager发出的各种事件
// Filter Graph Manager通过向视频窗口发送一个自定义消息WM_GRAPHNOTIFY来表示有事
// 件要处理
void CDXGraph::HandleEvent(WPARAM inWParam, LPARAM inLParam)
{
    if (mEvent)
    {
        LONG eventCode = 0, eventParam1 = 0, eventParam2 = 0;
        // 循环取出事件队列中的各个事件，分别作出处理
        while (SUCCEEDED(mEvent->GetEvent(&eventCode, &eventParam1,
            &eventParam2, 0)))
        {
            // 释放事件参数使用的资源
            mEvent->FreeEventParams(eventCode, eventParam1, eventParam2);
            switch (eventCode)
            {
            case EC_COMPLETE:   // 播放完成
            case EC_USERABORT:  // 用户中断
            case EC_ERRORABORT: // 错误中断
                Broadcast(msg_FilterGraphError);
                break;

            case EC_DEVICE_LOST:   // 设备丢失事件
                if (eventParam2 == 0)
                {
                    // 发送消息给上层监听者（控制对象）
                    Broadcast(msg_DShowDeviceLost, &eventParam1);
```

```
                }
                break;

            default:
                break;
            }
        }
    }
}

//
// CGraphController.cpp
//
// 控制对象上的自定义消息处理函数
bool CGraphController::ReceiveMessage(MessageT inMessage,  void * ioParam,
        void * ioParam2)
{
    switch (inMessage)
    {
    case msg_FilterGraphError:
        // 如果Filter Graph运行过程中发生错误，则将它停止
        StopGraph();
        return true;

    case msg_DShowDeviceLost:
        {
            // 判断丢失的是不是当前工作的设备
            IUnknown * pDevice = (IUnknown * ) (*((long*)ioParam));
            if (IsActiveDeviceLost(pDevice))
            {
                // 如果当前设备丢失，则停止Filter Graph的运行
                StopGraph();
            }
        }
        return true;
    }
    return CMsgReceiver::ReceiveMessage(inMessage, ioParam, ioParam2);
}
// 判断是不是当前工作的采集设备丢失
BOOL CGraphController::IsActiveDeviceLost(IUnknown * pLost)
{
    if (mInputFilters)
    {
        return mInputFilters->IsActiveDeviceLost(pLost);
    }
    return FALSE;
}

//
// CAnalogInputFilters.cpp
//
// 判断是不是当前工作的视频采集设备丢失
```

```
BOOL CAnalogInputFilters::IsActiveDeviceLost(IUnknown * pLost)
{
    if (mVideoCapture && mVideoCapture->GetFilter())
    {
        IUnknown * pDevice = NULL;
        mVideoCapture->GetFilter()->QueryInterface(IID_IUnknown,
            (void**)&pDevice);
        pDevice->Release();
        if (::IsEqualObject(pDevice, pLost))
        {
            return TRUE;
        }
    }

    return FALSE;
}

//
// CDVInputFilters.cpp
//
// 判断是不是当前工作的数码摄像机丢失
BOOL CDVInputFilters::IsActiveDeviceLost(IUnknown * pLost)
{
    if (mDVCapture && mDVCapture->GetFilter())
    {
        IUnknown * pDevice = NULL;
        mDVCapture->GetFilter()->QueryInterface(IID_IUnknown,
            (void**)&pDevice);
        pDevice->Release();
        if (::IsEqualObject(pDevice, pLost))
        {
            return TRUE;
        }
    }

    return FALSE;
}
```

2.2.9 支持即时抓图

所谓即时抓图，就是指在预览视频图像的时候，用户可以随时将看到的当前帧抓下来，保存为静态的BMP文件。

小知识：BMP 文件格式

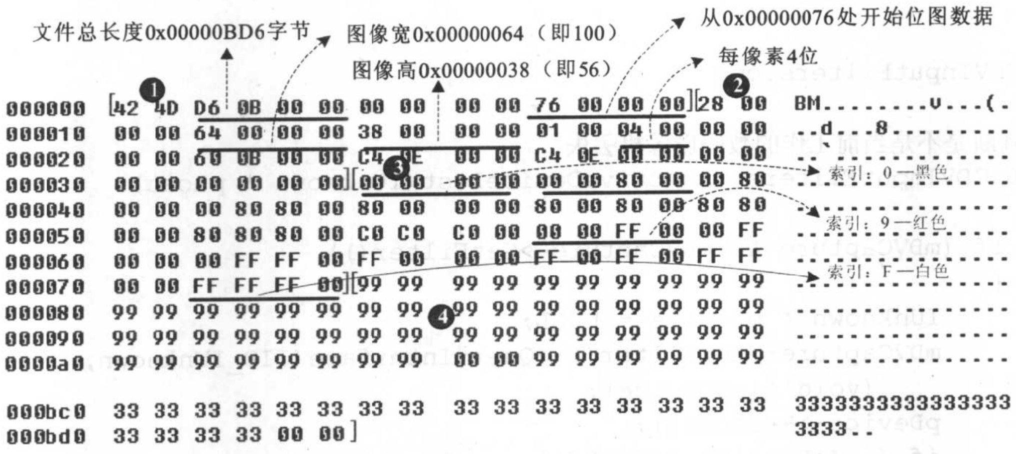
计算机上使用的静态图片文件有很多种格式，常见的有BMP、GIF、JPEG、PNG、PCX、TIFF等。其中，BMP（Bitmap的缩写）是一种比较简单的、在Windows上也用得最多的文件格式。

BMP文件可以用每像素1、4、8、16、24或32位来编码颜色信息，这个位数称为图像的颜色深度，它决定了图像所含的最大颜色数，如一幅1-bpp（bit per pixel，每个像素使用的

位数）的图像只能有两种颜色，而一幅24-bpp的图像可以有16777216（即2^{24}）种不同的颜色。颜色的数量越多，所表现的图像内容就越生动、越逼真。

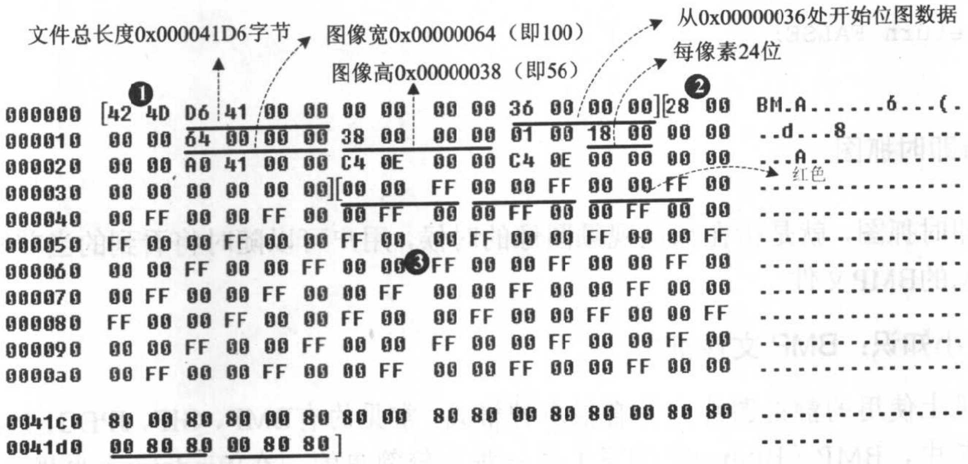
一个典型的BMP文件包含4个主要的部分：BMP文件头、BMP信息头、调色板和位图数据。如图2.17所示，一个100 x 56大小的位图，如果保存为16色的BMP文件，则它的文件数据如图2.18所示，如果保存为24位的BMP文件，则它的文件数据如图2.19所示。（注意：16、24、32位的位图均不使用调色板。）

图 2.17　演示用的位图



[❶] BITMAPFILEHEADER 数据结构（14字节）　[❷] BITMAPINFOHEADER 数据结构（40字节）
[❸] 调色板（定义16种颜色，每种颜色使用4字节）[❹]位图数据（每4位表示1个像素）

图 2.18　16 色 BMP 文件结构



[❶] BITMAPFILEHEADER数据结构（14字节）　[❷] BITMAPINFOHEADER数据结构（40字节）
[❸] 位图数据（每像素使用3字节：BGR BGR …）

图 2.19　24 位 BMP 文件结构

注意：GDI一般使用从下往上（Bottom-up）的方式来存储位图数据。从图2.18和图2.19中可以看出，位图数据中最先存放的是一系列红色的像素，也就是图2.17的底部图像。

1. BMP 文件头

BMP文件头使用一个数据结构BITMAPFILEHEADER来表示，它包含了BMP文件的类型、文件大小和位图数据的起始位置等信息。

```
typedef struct tagBITMAPFILEHEADER
{
   WORD    bfType;          // BMP文件的类型，必须为字符“BM”
   DWORD   bfSize;          // BMP文件的大小，以字节为单位
   WORD    bfReserved1;     // BMP文件保留字，必须为0
   WORD    bfReserved2;     // BMP文件保留字，必须为0
   DWORD   bfOffBits;       // 位图数据起始位置（相对于文件开头），以字节为单位
} BITMAPFILEHEADER;
```

2. BMP 信息头

BMP信息头使用一个数据结构BITMAPINFOHEADER来表示，它说明了位图的尺寸、颜色深度等信息。

```
typedef struct tagBITMAPINFOHEADER
{
   DWORD  biSize;              // 本结构所占用的字节数
   LONG   biWidth;             // 图像的宽度，以像素为单位
   LONG   biHeight;            // 图像的高度，以像素为单位
   WORD   biPlanes;            // 目标设备的平面数，必须为1
   WORD   biBitCount;          // 表示每个像素所需的位数
   DWORD  biCompression;       // 图像的压缩类型
   DWORD  biSizeImage;         // 图像的大小，以字节为单位
   LONG   biXPelsPerMeter;     // 图像水平分辨率，每米像素数
   LONG   biYPelsPerMeter;     // 图像垂直分辨率，每米像素数
   DWORD  biClrUsed;           // 位图实际使用的调色板中的颜色数
   DWORD  biClrImportant;      // 位图显示过程中重要的颜色数
} BITMAPINFOHEADER;
```

BITMAPINFOHEADER结构中一些重要成员的说明如下：

（1）biSize

本数据结构的大小为：

```
biSize = sizeof(BITMAPINFOHEADER);
```

（2）biWidth

图像的宽度，以像素为单位。如果位图数据是压缩的，则指定解压后的图像宽度。

（3）biHeight

图像的高度，以像素为单位。如果位图数据是压缩的，则指定解压后的图像高度。如果biHeight的值是正的，说明位图数据是从下往上（Bottom-up）扫描的，即位图数据中的

第一个像素对应图像最后一行左边的第一个像素，然后依次往右第二个像素、第三个像素……然后是图像的倒数第二行……直到图像的第一行。如果biHeight的值是负的，说明位图数据是从上往下（Top-down）扫描的，即位图数据依次对应图像的第一行、第二行……直到最后一行。图像的两种扫描方式如图2.20所示。

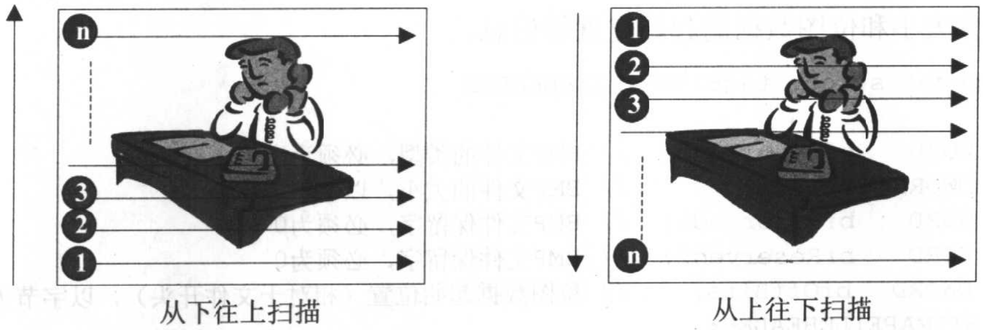


图 2.20　位图的两种扫描方式

（4）biBitCount

表示每个像素所需的位数，必须是0（可能是压缩数据）、1（2色）、4（16色）、8（256色）、16（最多2^{16}种颜色）、24（2^{24}种颜色）、32（最多2^{24}种颜色，多出来的8位用作Alpha通道值或者不用）。其中，1、4、8位的都需要调色板；16位时，如果biCompression成员的值为BI_RGB，则是RGB555的格式，如果biCompression成员的值为BI_BITFIELDS，则BITMAPINFOHEADER结构后面必须跟着用于提取RGB各分量的屏蔽字（3个DWORD类型的常量），图像的具体格式可能是RGB555，也可能是RGB565；24位时，每个像素使用3个字节表示，注意字节顺序是BGR；32位时，每个像素使用4个字节表示，注意字节顺序是BGRA，其中A是不用的字节或表示Alpha通道值。

（5）biSizeImage

图像的大小，以字节为单位，一般情况下它的计算公式为biSizeImage = biWidth x biHeight x biBitCount / 8。当图像的宽度字节数不是4的整数倍时，应以4为边界对齐，比如一幅16-bpp、宽高为11 x 11的位图，它的有效图像宽为11 x 16 / 8 = 22字节，而实际存储时应该对齐为24字节（多出的2个字节填充，显示时可以忽略），如图2.21所示。相应地，存储整幅图像需要的空间应该为24 x 11 = 264字节，而不是22 x 11 = 242字节。

因此，图像大小的正确计算方法应为：

```
long stride = (biWidth x biBitCount + 31) / 8;
biSizeImage = stride x biHeight;
```

（6）biCompression

位图的压缩格式可以是BI_RGB、BI_RLE8、BI_RLE4、BI_BITFIELDS、BI_JPEG、BI_PNG等。注意：如果位图数据是一种压缩格式，则这个位图一定是从下往上扫描的（从上往下扫描的位图不作压缩）。

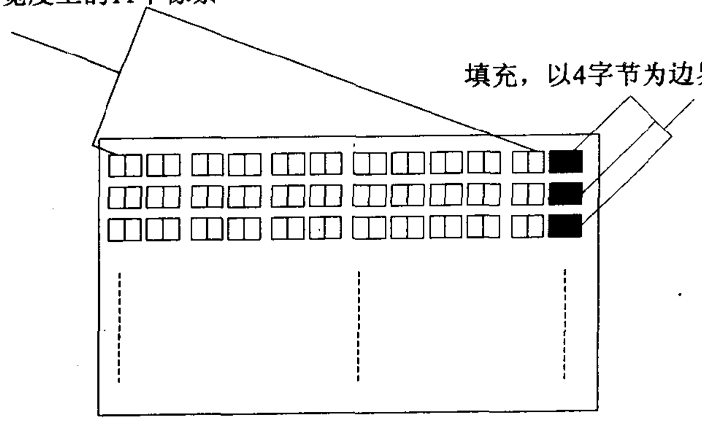


图2.21　16位、11 x 11位图的图像数据结构

3. 调色板

调色板用于说明位图中使用的颜色，它有若干个表项，每一个表项都用一个RGBQUAD数据结构（占用4个字节）定义一种颜色。RGBQUAD结构的定义如下：

```
typedef struct tagRGBQUAD {
  BYTE  rgbBlue;          // 蓝色分量(取值范围为0~255)
  BYTE  rgbGreen;         // 绿色分量(取值范围为0~255)
  BYTE  rgbRed;           // 红色分量(取值范围为0~255)
  BYTE  rgbReserved;      // 保留
} RGBQUAD;
```

调色板中RGBQUAD结构的个数由biBitCount的值来确定：当biBitCount等于1、4、8时，分别有2、16、256个表项；当biBitCount等于16、24、32时，表项个数为0（也就是说不使用调色板）。BMP信息头和调色板组成位图信息——另一个数据结构BITMAPINFO，定义如下：

```
typedef struct tagBITMAPINFO {
   BITMAPINFOHEADER bmiHeader;      // BMP信息头
   RGBQUAD          bmiColors[1];   // 调色板颜色数组
} BITMAPINFO;
```

4. 位图数据

位图数据记录了位图的每一个像素值，记录顺序为：在扫描行内从左到右，而在扫描行之间从下往上（biHeight值为正数）或从上往下（biHeight值为负数）。位图的一个像素值所占的字节数由biBitCount的值确定。值得注意的是，当biBitCount等于1、4、8时，位图数据中的各像素值表示的不是真正的颜色值，而是颜色值在调色板中对应的索引值；当biBitCount等于16、24、32时，位图数据中的各像素值才直接表示这个像素的RGB分量值。

实现即时抓图，最简单的方法是使用Filter Graph Manager上的IBasicVideo接口（该接口真正实现在Video Renderer上），调用这个接口的GetCurrentImage函数就可以将当前帧抓下来。

```
//
// CDXGraph.cpp
//
// 调用IBasicVideo::GetCurrentImage抓取当前播放时刻的图像帧
BOOL CDXGraph::GetCurrentImage(const char * inFile)
{
    // mBasicVideo为Filter Graph Manager上获得的IBasicVideo接口对象指针
    if (!mBasicVideo)
    {
        return FALSE;
    }

    long bitmapSize = 0;
    // 第一次调用GetCurrentImage返回读取一帧需要的缓存大小
    if (SUCCEEDED(mBasicVideo->GetCurrentImage(&bitmapSize, 0)))
    {
        BOOL pass = FALSE;
        // 申请足够大的缓存用于装载图像数据
        BYTE * buffer = new BYTE[bitmapSize];
        // 第二次调用GetCurrentImage返回图像数据（包括BMP信息头）
        if (SUCCEEDED(mBasicVideo->GetCurrentImage(&bitmapSize, (long
            *)buffer)))
        {
            BITMAPFILEHEADER    hdr;    // BMP文件头
            LPBITMAPINFOHEADER  lpbi;   // BMP信息头指针

            lpbi = (LPBITMAPINFOHEADER)buffer;
            int nColors = 1 << lpbi->biBitCount; // 颜色数
            // 如果颜色数超过256，则该图像不使用调色板
            if (nColors > 256)
            {
                nColors = 0;
            }
            // 初始化BMP文件头
            hdr.bfType    = ((WORD) ('M' << 8) | 'B'); // always is "BM"
            hdr.bfSize    = bitmapSize + sizeof( hdr );
            hdr.bfReserved1 = 0;
            hdr.bfReserved2 = 0;
            hdr.bfOffBits  = (DWORD) (sizeof(BITMAPFILEHEADER) +
                lpbi->biSize + nColors * sizeof(RGBQUAD));

            // 保存为一个BMP文件
            CFile bmpFile(inFile, CFile::modeReadWrite | CFile::modeCreate
                | CFile::typeBinary);
            bmpFile.Write(&hdr, sizeof(BITMAPFILEHEADER));
            bmpFile.Write(buffer, bitmapSize);
            bmpFile.Close();
            pass = TRUE;
        }
        delete [] buffer;
        return pass;
    }
```

```
    return TRUE;
}
```

遗憾的是，在传统的Video Renderer上实现的IBasicVideo::GetCurrentImage并不稳定（如果使用VMR则没有这个问题）。那怎么办呢？办法当然是有的。大家知道，如果自己写一个Transform Filter（这个Filter名字暂定为HQ Image Grabber）接入到采集设备Filter后面的链路中，则Filter Graph中流动的每一个Sample（也就是图像帧）都会经过我们的Filter。只要在我们的Filter上实现一个自定义接口，用于将当前时刻的Sample数据拷贝出来，也就是实现了即时抓图。

当处理非AGP接口的采集设备时，可以将HQ Image Grabber接到Preview输出Pin后面（如果没有Preview输出Pin则接到Capture输出Pin后面），如图2.22所示。

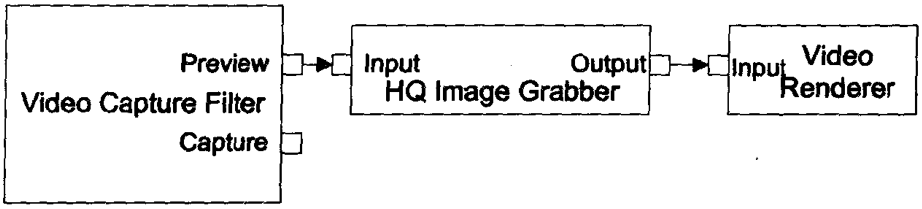


图 2.22　支持对非 AGP 接口的采集设备即时抓图

数码摄像机的情况与上述情况类似，如图2.23所示。

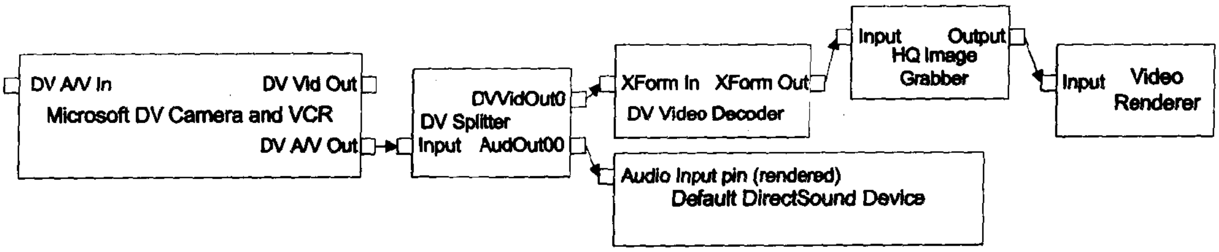


图 2.23　支持对数码摄像机即时抓图

处理AGP接口的采集设备有点麻烦。因为AGP采集设备在预览时，VP Pin根本不输出图像数据。于是，我们只能从它的Capture输出Pin上打主意：将HQ Image Grabber接到Capture输出Pin上，然后要再接一个Renderer Filter以便构成一条封闭的链路。接什么Renderer Filter呢？因为此时不再需要显示Capture Pin输出的图像，所以可以选择Null Renderer（一个什么也不做的Renderer Filter）。构建的Filter Graph如图2.24所示。

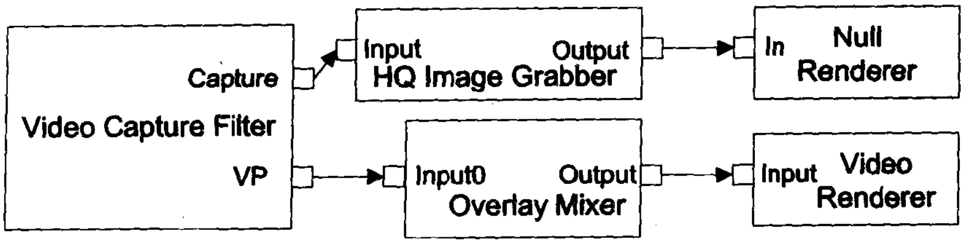


图 2.24　支持对 AGP 接口的采集设备即时抓图

提示：为简单起见，AVCap程序只在预览模式下支持即时抓图。有兴趣的读者可以自行扩展，使其在采集模式下也能进行即时抓图。

具体实现时，首先要开发一个HQ Image Grabber Filter。因为这个Filter只需得到访问Sample的机会，所以可以从CTransInPlaceFilter类继承，并且实现一个自定义接口IImageGrabber。

```
//
// IImageGrabber.h
//

#ifndef __H_IImageGrabber__
#define __H_IImageGrabber__

#ifdef __cplusplus
extern "C" {
#endif

//----------------------------------------------------------------------
// IImageGrabber接口定义
//----------------------------------------------------------------------
DECLARE_INTERFACE_(IImageGrabber, IUnknown)
{
    // 得到图像的宽、高、每像素的位数
    STDMETHOD(get_ImageSize) (THIS_
        long * outWidth, long * outHeight, long * outBitCount
    ) PURE;
    // 得到图像帧的大小
    STDMETHOD(get_FrameSize) (THIS_
        long * outFrameSize
    ) PURE;
    // 得到BMP信息头
    STDMETHOD(get_BitmapInfoHeader) (THIS_
        BITMAPINFOHEADER * outBitmapInfo
    ) PURE;
    // 判断输入媒体类型是不是16位的RGB，是RGB555还是RGB565
    STDMETHOD(get_Is16BitsRGB) (THIS_
        long * outIsRGB565
    ) PURE;
    // 设置标记：输入图像是否是场图像
    STDMETHOD(put_IsFieldPicture) (THIS_
        BOOL inIsField
    ) PURE;
    // 抓取当前帧图像，可以有同步和异步两种模式
    STDMETHOD(Snapshot) (THIS_
        BYTE * outBuffer, BOOL inIsSyncMode
    ) PURE;
    // 判断异步模式下抓图操作是否结束
    STDMETHOD(IsAsyncSnapshotFinished) (THIS_
        BOOL * outFinished
    ) PURE;
```

```
    // 取消所有等候的操作（主要用于异步抓图模式）
    STDMETHOD(CancelAllPending) (THIS) PURE;
};
//-------------------------------------------------------------------------

#ifdef __cplusplus
}
#endif

#endif // __H_IImageGrabber__

//
// CFilterImageGrabber.h
//

#ifndef __H_CFilterImageGrabber__
#define __H_CFilterImageGrabber__

#include "IImageGrabber.h"

class CFilterImageGrabber : public CTransInPlaceFilter, public
    IImageGrabber
{
private:
    CFilterImageGrabber(TCHAR *tszName, LPUNKNOWN punk, HRESULT *phr);
    ~CFilterImageGrabber();

    CCritSec    mSafeCapture;       // 同步对象
    BYTE *      mFrameBuffer;       // 图像帧缓存
    BOOL        mIsFieldPicture;    // 场图像标记
    long        mImageWidth;        // 图像的宽
    long        mImageHeight;       // 图像的高
    long        mImageSize;         // 图像帧大小
public:
    static CUnknown * WINAPI CreateInstance(LPUNKNOWN punk, HRESULT *phr);
    DECLARE_IUNKNOWN;
    STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void ** ppv);
    // 实现必要的Filter框架函数
    virtual HRESULT CheckInputType(const CMediaType* mtIn); // PURE
    virtual HRESULT Transform(IMediaSample *pSample); // PURE
    HRESULT CompleteConnect(PIN_DIRECTION dir,IPin *pReceivePin);
    HRESULT StartStreaming(void);
    // 如果输入的是场图像，则移掉一场
    void VerifyFieldPicture(BYTE * pImage);
    // --- IImageGrabber的各个接口方法 ---
    STDMETHODIMP get_ImageSize(long * outWidth, long * outHeight, long *
        outBitCount);
    STDMETHODIMP get_FrameSize(long * outFrameSize);
    STDMETHODIMP get_BitmapInfoHeader(BITMAPINFOHEADER * outBitmapInfo);
    STDMETHODIMP get_Is16BitsRGB(long * outIsRGB565);
    STDMETHODIMP put_IsFieldPicture(BOOL inIsField);
    STDMETHODIMP Snapshot(BYTE * outBuffer, BOOL inIsSyncMode);
```

```
    STDMETHODIMP IsAsyncSnapshotFinished(BOOL * outFinished);
    STDMETHODIMP CancelAllPending();
};

#endif // __H_CFilterImageGrabber__

//
// CFilterImageGrabber.cpp
//
// 在Filter上暴露IImageGrabber接口
STDMETHODIMP CFilterImageGrabber::NonDelegatingQueryInterface(REFIID
    riid, void ** ppv)
{
    CheckPointer(ppv, E_POINTER);
    if (riid == IID_IImageGrabber)
    {
        return GetInterface((IImageGrabber *) this, ppv);
    }
    else
    {
        return CTransInPlaceFilter::NonDelegatingQueryInterface(riid,
            ppv);
    }
}
// 检查输入的媒体类型
HRESULT CFilterImageGrabber::CheckInputType(const CMediaType* mtIn)
{
    // 不允许动态改变媒体类型
    if (m_State != State_Stopped)
    {
        return E_FAIL;
    }

    // 仅支持16位、24位、32位的RGB媒体类型
    if (mtIn->majortype == MEDIATYPE_Video)
    {
        if (mtIn->subtype == MEDIASUBTYPE_RGB32 ||
            mtIn->subtype == MEDIASUBTYPE_RGB24 ||
            mtIn->subtype == MEDIASUBTYPE_RGB565 ||
            mtIn->subtype == MEDIASUBTYPE_RGB555)
        {
            return NOERROR;
        }
    }
    return E_FAIL;
}
// 当输入Pin完成连接时，取得输入的媒体信息
HRESULT CFilterImageGrabber::CompleteConnect(PIN_DIRECTION dir,IPin
    *pReceivePin)
{
    HRESULT hr = CTransInPlaceFilter::CompleteConnect(dir, pReceivePin);
    if (SUCCEEDED(hr) && dir == PINDIR_INPUT)
```

```
    {
        CMediaType  mtIn = m_pInput->CurrentMediaType();
        // 从媒体类型中取得输入图像的宽、高、图像帧大小等信息
        if (mtIn.formattype == FORMAT_VideoInfo)
        {
            VIDEOINFOHEADER * pInfo = (VIDEOINFOHEADER *) mtIn.pbFormat;
            mImageWidth  = pInfo->bmiHeader.biWidth;
            mImageHeight = abs(pInfo->bmiHeader.biHeight);
            mImageSize = mImageWidth * mImageHeight *
                pInfo->bmiHeader.biBitCount / 8;
        }
        else if (mtIn.formattype == FORMAT_VideoInfo2)
        {
            VIDEOINFOHEADER2 * pInfo = (VIDEOINFOHEADER2 *) mtIn.pbFormat;
            mImageWidth  = pInfo->bmiHeader.biWidth;
            mImageHeight = abs(pInfo->bmiHeader.biHeight);
            mImageSize = mImageWidth * mImageHeight *
                pInfo->bmiHeader.biBitCount / 8;
        }
    }
    return hr;
}
// 输入Pin上接收到的Sample将被传送到这个Transform函数中
// 如果mFrameBuffer指针非空，则抓取Sample的数据到这个缓存中
HRESULT CFilterImageGrabber::Transform(IMediaSample *pSample)
{
    // mFrameBuffer是外部调用者提供的帧缓存地址
    if (mFrameBuffer)
    {
        BYTE * pData;
        pSample->GetPointer(&pData);
        {
            CAutoLock lck(&mSafeCapture); // 锁定!
            // 将Sample数据（当前图像帧）拷贝出来
            CopyMemory(mFrameBuffer, pData, mImageSize);
            // 修正场图像数据
            VerifyFieldPicture(mFrameBuffer);
            // 将mFrameBuffer赋值为NULL表示当前抓帧完成
            mFrameBuffer = NULL;
        }
    }
    return NOERROR;
}
// 如果输入图像是场图像，则移掉一场，以提高图像的清晰度
void CFilterImageGrabber::VerifyFieldPicture(BYTE * pImage)
{
    // 如果输入图像的宽度不大于360，一般不会是场图像，因此仅对图像宽度大于360的场图像
    // 进行特殊处理
    if (mIsFieldPicture && mImageHeight > 360)
    {
        // 图像的宽度（以字节为单位）
        long lineLength  = mImageSize / mImageHeight;
```

```
            // 2倍的图像宽度（用于相邻奇数行之间或相邻偶数行之间的跳转）
            long lineLength2 = (lineLength << 1);
            // 图像的一半高度（因为奇场和偶场图像各占整幅图像的一半大小）
            int  halfHeight  = mImageHeight / 2;
            // 移掉一场图像：以奇数行数据覆盖偶数行数据
            BYTE * currentPtr = pImage;
            for (int i = 0; i < halfHeight; i++)
            {
                ::memcpy(currentPtr + lineLength, currentPtr, lineLength);
                // 指向下一个奇数行
                currentPtr += lineLength2;
            }
        }
    }

// --- IImageGrabber各接口方法的实现 ---
// 得到图像的宽、高、每像素的位数（从输入Pin上连接用的媒体类型上获取）
STDMETHODIMP CFilterImageGrabber::get_ImageSize(long * outWidth, long *
    outHeight, long * outBitCount)
{
    // 首先判断输入Pin是否已经连接
    if (m_pInput->IsConnected())
    {
        // 获得输入Pin上当前连接用的媒体类型
        CMediaType mt = m_pInput->CurrentMediaType();
        if (mt.formattype == FORMAT_VideoInfo)
        {
            VIDEOINFOHEADER * pInfo = (VIDEOINFOHEADER *) mt.Format();
            *outWidth  = pInfo->bmiHeader.biWidth;
            *outHeight = abs(pInfo->bmiHeader.biHeight);
            *outBitCount = pInfo->bmiHeader.biBitCount;
        }
        else if (mt.formattype == FORMAT_VideoInfo2)
        {
            VIDEOINFOHEADER2 * pInfo = (VIDEOINFOHEADER2 *) mt.Format();
            *outWidth  = pInfo->bmiHeader.biWidth;
            *outHeight = abs(pInfo->bmiHeader.biHeight);
            *outBitCount = pInfo->bmiHeader.biBitCount;
        }
        return NOERROR;
    }
    return E_FAIL;
}
// 得到图像帧的大小
STDMETHODIMP CFilterImageGrabber::get_FrameSize(long * outFrameSize)
{
    *outFrameSize = mImageSize;
    return NOERROR;
}
// 得到BMP信息头（从输入Pin上连接用的媒体类型上获取）
STDMETHODIMP CFilterImageGrabber::get_BitmapInfoHeader(BITMAPINFOHEADER *
    outBitmapInfo)
```

```
{
    // 首先判断输入Pin是否已经连接
    if (m_pInput->IsConnected())
    {
        // 获得输入Pin上当前连接用的媒体类型
        CMediaType mt = m_pInput->CurrentMediaType();
        if (mt.formattype == FORMAT_VideoInfo)
        {
            VIDEOINFOHEADER * pInfo = (VIDEOINFOHEADER *) mt.Format();
            *outBitmapInfo = pInfo->bmiHeader;
        }
        else if (mt.formattype == FORMAT_VideoInfo2)
        {
            VIDEOINFOHEADER2 * pInfo = (VIDEOINFOHEADER2 *) mt.Format();
            *outBitmapInfo = pInfo->bmiHeader;
        }
        return NOERROR;
    }
    return E_FAIL;
}
// 判断输入媒体类型是不是16位的RGB，是RGB555还是RGB565
STDMETHODIMP CFilterImageGrabber::get_Is16BitsRGB(long * outIsRGB565)
{
    *outIsRGB565 = 0;
    // 首先判断输入Pin是否已经连接
    if (m_pInput->IsConnected())
    {
        // 获得输入Pin上当前连接用的媒体类型
        CMediaType mt = m_pInput->CurrentMediaType();
        if (mt.subtype == MEDIASUBTYPE_RGB565)
        {
            *outIsRGB565 = 1;
        }
        else if (mt.subtype == MEDIASUBTYPE_RGB555)
        {
            *outIsRGB565 = 2;
        }
    }
    return NOERROR;
}
// 设置标记：输入图像是否是场图像
STDMETHODIMP CFilterImageGrabber::put_IsFieldPicture(BOOL inIsField)
{
    mIsFieldPicture = (BOOL) inIsField;
    return NOERROR;
}
// 抓取当前帧图像，可以有同步和异步两种模式（实际上抓取的是下一个进来的Sample）
STDMETHODIMP CFilterImageGrabber::Snapshot(BYTE * outBuffer, BOOL
    inIsSyncMode)
{
    ASSERT(outBuffer);
    // Filter和Pin的状态检查
```

```
    if (!m_pInput->IsConnected() || !m_pOutput->IsConnected() ||
        m_State != State_Running)
    {
        return E_FAIL;
    }

    {
        CAutoLock lck(&mSafeCapture);
        mFrameBuffer = outBuffer;
    }
    // 如果是同步模式，必须等到抓帧完成后才返回
    while (mFrameBuffer && inIsSyncMode)
    {
        if (m_State != State_Running)
        {
            break;
        }

        Sleep(5);
    }
    return NOERROR;
}
// 判断异步模式下抓图操作是否结束
STDMETHODIMP CFilterImageGrabber::IsAsyncSnapshotFinished(BOOL *
    outFinished)
{
    *outFinished = (mFrameBuffer == NULL);
    return NOERROR;
}
// 取消所有等候的操作（主要用于异步抓图模式）
STDMETHODIMP CFilterImageGrabber::CancelAllPending()
{
    {
        CAutoLock lck(&mSafeCapture);
        mFrameBuffer = NULL;
    }
    return NOERROR;
}
```

提示：本书配套光盘的SourceCodes\Chapter02\FilterImageGrabber目录下提供了HQ Image Grabber Filter的完整实现。

小知识：帧图像与场图像

数码摄像机（或照相机）摄像时，内部CCD传感器读取数据的方式通常有两种：逐行扫描和隔行扫描。前者在拍摄时一次性将整幅画面读取下来，也就获得了一帧图像；而后者将一幅画面水平分成间隔的像素行，通过电子快门，每次轮换读取奇数行和偶数行的信号（如第一次同时读取1、3、5、7……行的数据，第二次同时读取2、4、6、8……行的数据）。隔行扫描每次读取的数据称为场图像，两个场图像组成一帧图像。

逐行扫描获取图像的方式可以避免运动物体的模糊，数码照相机大都采用这种方式。

隔行扫描两次信号读取的时间差较小，为运动视频的重放带来了更好的流畅感，数码摄像机大都采用这种方式。但是，如果使用数码摄像机来拍摄单帧图像（特别是画面中有运动物体的场景），则两场信号获取的时间差就会导致明显的画面差异，奇偶场合并组成的图像在清晰度方面表现较差。

对于即时抓图应用，如果信号源自数码摄像机，当视频图像的尺寸为720 x 480或720 x 576时，这时候的每一帧图像都由两场组成，并且是交替存储的，如奇1-偶2-奇3-偶4-奇5-偶6……为了得到较高清晰度的图像，应将其中一场图像移掉，输出一半高度的图像，如奇1-奇3-奇5……或偶2-偶4-偶6……也可以保持原图像高度不变，将其中一场的数据覆盖另外一场，如奇1-奇1-奇3-奇3-奇5-奇5……或偶2-偶2-偶4-偶4-偶6-偶6……

HQ Image Grabber Filter开发完成之后，就可以将它应用到AVCap程序中去了。首先给它提供一个包装类CImageGrabberFilter，然后就是在控制类CPreviewController中将其创建、并在连接Filter的时候接入。

```
//
// CPreviewController.cpp
//
// 完成预览用的输入部分与输出部分的Filter连接
BOOL CPreviewController::ConnectFilters(void)
{
    BOOL pass   = FALSE;
    IPin * pOut = NULL;
    IPin * pIn  = NULL;

    // 如果Image Grabber Filter创建了，则将其接入到链路中
    if (mGrabber && mGrabber->GetFilter())
    {
        IPin * snapshotPin = NULL;
        if (mInputFilters->IsAGPCard())  // 如果从AGP设备输入
        {
            // 首先在Capture输出Pin上连接Image Grabber Filter
            snapshotPin = mInputFilters->GetVideoOutPin();
            pass = TryToConnectImageGrabber(snapshotPin);
            // 然后在Image Grabber Filter上连接Null Renderer
            pOut = mGrabber->GetPin(FALSE);
            if (!mNullRenderer)
            {
                mNullRenderer = new CDXFilter(mGraph->GetGraph(),
                    CLSID_NullRenderer, "Null Renderer");
                mNullRenderer->CreateFilter();
            }
            pIn  = mNullRenderer->GetPin(TRUE);
            pass = mGraph->ConnectFilters(pOut, pIn);
            // 获取AGP设备的Preview输出Pin (用于连到Video Renderer)
            pOut = mInputFilters->GetPreviewPin();
        }
        else    // 如果从非AGP设备输入
        {
            // 在Preveiw输出Pin后面首先连接Image Grabber Filter
```

```
            snapshotPin = mInputFilters->GetPreviewPin();
            pass = TryToConnectImageGrabber(snapshotPin);
            // 获取Image Grabber Filter的输出Pin（用于连到Video Renderer）
            pOut = mGrabber->GetPin(FALSE);
        }
    }
    else
    {
        // 如果不需要支持即时抓图，则直接取出Preview输出Pin
        pOut = mInputFilters->GetPreviewPin();
    }
    // 最后将视频链路连接到Video Renderer上
    pIn = mVideoRenderer->GetInputPin();
    if (pOut && pIn)
    {
        pass = mGraph->ConnectFilters(pOut, pIn);
    }

    // 连接输入部分的音频输出Pin到Audio Renderer
    pOut = mInputFilters->GetAudioOutPin();
    pIn  = mAudioRenderer->GetPin(TRUE);
    if (pass && pOut && pIn)
    {
        pass = mGraph->ConnectFilters(pOut, pIn);
    }
    return pass;
}
// 将Image Grabber Filter接入Filter链路
BOOL CPreviewController::TryToConnectImageGrabber(IPin * inOutputPin)
{
    IPin * pOut = inOutputPin;
    IPin * pIn  = mGrabber->GetPin(TRUE);
    // 首先尝试直接连接Image Grabber Filter
    BOOL pass = mGraph->ConnectFilters(pOut, pIn);
    if (!pass)
    {
        // 如果直接连接不成功（因为Image Grabber Filter只支持RGB输入），则首先接
        // 入一个AVI Decompressor，将inOutputPin的媒体类型转化成RGB
        if (!mAviDecoder)
        {
            mAviDecoder = new CDXFilter(mGraph->GetGraph(),
                CLSID_AVIDec, "AVI Decompressor");
            mAviDecoder->CreateFilter();
        }
        pIn = mAviDecoder->GetPin(TRUE);
        pass = mGraph->ConnectFilters(pOut, pIn);
        // 最后将AVI Decompressor的输出Pin连接到Image Grabber Filter
        pOut = mAviDecoder->GetPin(FALSE);
        pIn  = mGrabber->GetPin(TRUE);
        pass = mGraph->ConnectFilters(pOut, pIn);
    }
    return pass;
```

```
}
// 创建预览用的输出部分的Filter
BOOL CPreviewController::CreateOutputFilters(void)
{
    BOOL pass = CGraphController::CreateOutputFilters();
    if (pass)
    {
        // 如果参数对象中关于是否支持即时抓图的标记有效，则创建一个Image Grabber
        // Filter
        if (mLiveCapture->IsGrabEnabled())
        {
            mGrabber = new CImageGrabberFilter(mGraph->GetGraph());
            pass = mGrabber->CreateFilter();
            // 信号源输出的图像是否为场图像（数码摄像机是，采集卡不是）
            mGrabber->SetFieldPicture(mInputFilters->IsFieldPicture());
        }
    }
    return pass;
}
```

用户在通过AVCap程序主界面上的Grab按钮进行即时抓图时，程序的响应流程如下：

```
//
// AVCapDlg.cpp
//
// AVCap程序主界面上的Grab按钮的响应函数
void CAVCapDlg::OnButtonGrab()
{
    if (mController)
    {
        // 获取当前系统时间
        CTime   currentTime = CTime::GetCurrentTime();
        // 生成文件名字符串
        CString  filename;
        filename.Format("%shq%02d-%02d-%02d.bmp",
            mLiveCapture.GetGrabFolder(),currentTime.GetHour(),
            currentTime.GetMinute(),currentTime.GetSecond());
        // 调用控制对象的即时抓图函数
        // 即时抓图完成后弹出操作成功与否的消息框
        if (mController->SnapshotToFile(filename))
        {
            AfxMessageBox("Grab to "+filename+" succeeded!");
        }
        else
        {
            AfxMessageBox("Failed to capture a frame!");
        }
    }
}

//
// CPreviewController.cpp
```

```
//
// 控制对象上的即时抓图函数
BOOL CPreviewController::SnapshotToFile(const char * inFile)
{
    if (mIsGraphActive) // 当前Filter Graph是否构建成功?
    {
        // 如果Image Grabber Filter完全接入Filter链路了，则调用该Filter上的接口
        // 函数进行当前帧捕获
        if (mGrabber && mGrabber->GetFilter() &&
            UFilterUtils::IsCompletelyConnected(mGrabber->GetFilter()))
        {
            return mGrabber->SnapshotToFile(inFile);
        }
        else if (mLiveCapture->IsVMRPreferred())
        {
            // 如果使用了VMR，可以放心使用IBasicVideo::GetCurrentImage
            return mGraph->GetCurrentImage(inFile);
        }
    }
    return FALSE;
}

//
// UFilterUtils.cpp
//
// 判断指定的Filter是否完全连接，即所有的输入Pin和输出Pin都已经完成连接
BOOL UFilterUtils::IsCompletelyConnected(IBaseFilter * inFilter)
{
    if (!inFilter)
    {
        return FALSE;
    }

    BOOL allConnected  = TRUE;
    IEnumPins * pinEnum = NULL;
    // 枚举Filter上所有的Pin
    if (SUCCEEDED(inFilter->EnumPins(&pinEnum)))
    {
        pinEnum->Reset();

        IPin * pin = NULL;
        ULONG fetchCount = 0;
        while (SUCCEEDED(pinEnum->Next(1, &pin, &fetchCount)) &&
            fetchCount && allConnected)
        {
            if (pin)
            {
                pin->Release();
                // 判断当前Pin是否已经连接到另外一个Pin
                IPin * connected = NULL;
                pin->ConnectedTo(&connected);
                if (connected)
```

```
                {
                    connected->Release();
                }
                else
                {
                    allConnected = FALSE;
                }
            }
        }
        pinEnum->Release();
    }
    return allConnected;
}

//
// CImageGrabberFilter.cpp
//
//#define _ASYNC_SNAPSHOT
// 即时抓图保存到文件inFile：演示了IImageGrabber各接口函数的使用方法
// mIGrabber为IImageGrabber接口对象指针，在CreateFilter函数调用时获得
BOOL CImageGrabberFilter::SnapshotToFile(const char * inFile)
{
    // 获取图像帧的大小，以便申请足够大的缓存
    long frameSize = 0;
    if (GetFrameSize(&frameSize))
    {
        BYTE * pBuffer = new BYTE[frameSize];

#ifndef _ASYNC_SNAPSHOT
        // --- 同步模式抓图 ---
        // Snapshot函数的第二个参数为TRUE表示同步模式
        if (Snapshot(pBuffer, TRUE))
        {
            // 将抓到的图像数据保存为BMP文件
            WriteImageFile(pBuffer, frameSize, inFile);
        }
#else
        // --- 异步模式抓图 ---
        // Snapshot函数的第二个参数为FALSE表示异步模式
        Snapshot(pBuffer, FALSE);
        long counter = 0;
        // 循环判断抓图操作是否完成，如果没完成就等待，但等待时间最长不超过2秒
        while (!IsAsyncSnapshotFinished())
        {
            Sleep(2);
            if (++counter > 1000)
            {
                break;
            }
        }
        if (counter > 1000)
        {
```

```
            // 放弃当前的即时抓图操作
            CancelAllPending();
        }
        else
        {
            // 将抓到的图像数据保存为BMP文件
            WriteImageFile(pBuffer, frameSize, inFile);
        }
#endif
        // 释放图像帧缓存
        if (pBuffer)
        {
            delete[] pBuffer;
            pBuffer = NULL;
        }
        return TRUE;
    }
    return FALSE;
}
// 获取输入图像帧的大小
BOOL CImageGrabberFilter::GetFrameSize(long * outFrameSize)
{
    if (mIGrabber)
    {
        HRESULT hr = mIGrabber->get_FrameSize(outFrameSize);
        return SUCCEEDED(hr);
    }
    return FALSE;
}
// 得到当前图像帧的数据
BOOL CImageGrabberFilter::Snapshot(BYTE * outBuffer, BOOL inIsSyncMode)
{
    if (mIGrabber)
    {
        HRESULT hr = mIGrabber->Snapshot(outBuffer, inIsSyncMode);
        return SUCCEEDED(hr);
    }
    return FALSE;
}

// 其他接口函数的包装函数（省略）
// ……

// 将图像数据保存为BMP文件
void CImageGrabberFilter::WriteImageFile(BYTE * inData, long inSize,const
    char * inFile)
{
    BITMAPINFOHEADER info;
    GetBitmapInfoHeader(&info);
    int nColors = 1 << info.biBitCount;
```

```
    if (nColors > 256)
    {
        nColors = 0;
    }

    long isRGB565 = 0;
    // 判断输入图像是不是16位的RGB格式，是RGB555还是RGB565
    // 如果是，则要在BMP信息头后面跟3个DWORD类型的屏蔽字
    BOOL is16Bits = Is16BitsRGB(&isRGB565);
    const DWORD bits555[] = {0x7C00,0x03E0,0x001F};
    const DWORD bits565[] = {0xF800,0x07E0,0x001F};
    // 定义BMP文件头
    BITMAPFILEHEADER    hdr;
    hdr.bfType      = ((WORD) ('M' << 8) | 'B');    // always is "BM"
    hdr.bfSize      = inSize + sizeof(hdr) + sizeof(BITMAPINFOHEADER);
    hdr.bfReserved1 = 0;
    hdr.bfReserved2 = 0;
    hdr.bfOffBits   = (DWORD) (sizeof(BITMAPFILEHEADER) + info.biSize +
        nColors * sizeof(RGBQUAD));
    // 如果是16位RGB格式，需要修正如下成员（因为加入了3个屏蔽字）
    if (is16Bits)
    {
        hdr.bfSize    += 12;
        hdr.bfOffBits += 12;
    }

#ifdef _DEBUG
    int offset1 = sizeof(BITMAPFILEHEADER);
    int offset2 = sizeof(BITMAPINFOHEADER);
#endif

    FILE * fp = fopen(inFile, "wb");
    if (fp != NULL)
    {
        // 写入BMP文件头
        fwrite(&hdr, sizeof(char), sizeof(BITMAPFILEHEADER), fp);
        // 写入BMP信息头
        fwrite(&info, sizeof(char), sizeof(BITMAPINFOHEADER), fp);
        // 因为Image Grabber Filter不支持带调色板的RGB格式输入，所以这里不需要向
        // 文件中写入调色板数据
        // ……
        if (is16Bits)
        {
            // 写入3个DWORD类型的屏蔽字
            if (isRGB565 == 1) // RGB565
            {
                fwrite(bits565, sizeof(DWORD), 3, fp);
            }
            else  // RGB555
```

```
            {
                fwrite(bits555, sizeof(DWORD), 3, fp);
            }
        }
        // 写入图像数据
        fwrite(inData, sizeof(char), inSize, fp);
        fclose(fp);
    }
}
```

2.2.10 采集到AVI文件

当视频预览功能实现之后，更进一步的需求随即提上了开发日程：如何将视频内容采集到文件中保存？AVCap设计的程序框架，使从单纯预览功能的实现扩展到支持采集功能变得异常容易，有一种“顺水行舟”的感觉！

非AGP接口的采集设备构建采集用的Filter Graph如图2.25所示（采集的同时也有预览）。

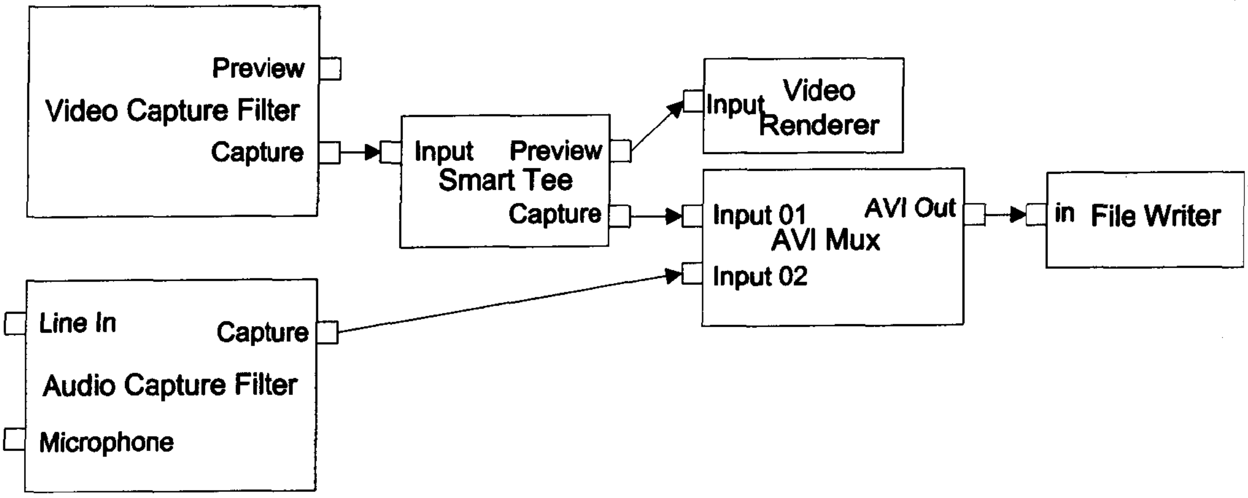


图 2.25 AGP 接口的采集设备采集用的 Filter Graph

提示： 为什么对非AGP接口的采集设备在实现采集时的预览时，不直接从采集设备Filter的Preview输出Pin接出，而要在Capture输出Pin后面接一个Smart Tee呢？大家知道，PCI总线的带宽是133MB/s，并且是给系统中所有的PCI设备共享的。假设采集输出的图像大小为720 x 576，每个像素占用3个字节，视频帧率为25fps，则一路采集输出将占用的带宽为720 x 576 x 3 x 25 / 1024 / 1024 ≈ 29.66MB/s，两路将是59.32MB/s。如果Preview输出Pin和Capture输出Pin同时使用，可能会消耗掉PCI总线将近一半的带宽，严重时可能会影响计算机系统的性能。如图2.25构建的Filter Graph同样能够实现采集时的预览功能，但占用的系统资源要少很多。

AGP接口的采集设备构建采集用的Filter Graph如图2.26所示。

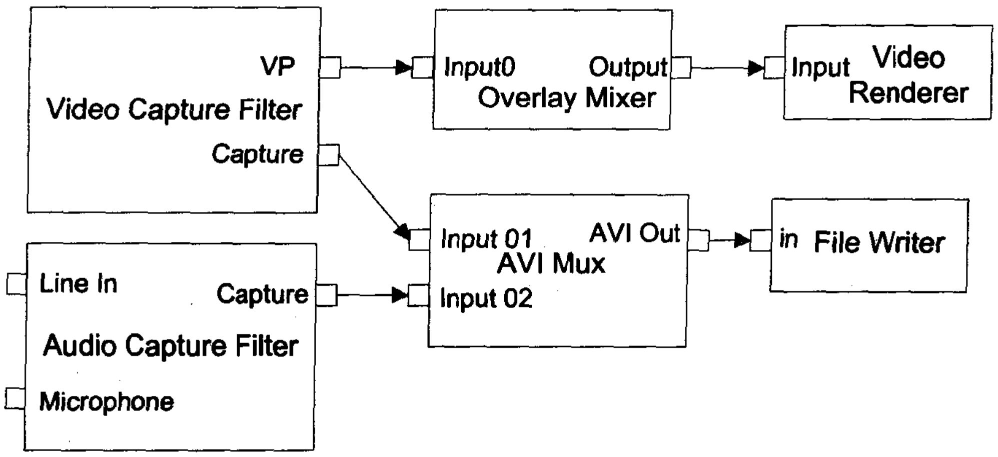


图 2.26　AGP 接口的采集设备采集用的 Filter Graph

如果信号源自数码摄像机，构建Filter Graph如图2.27所示。

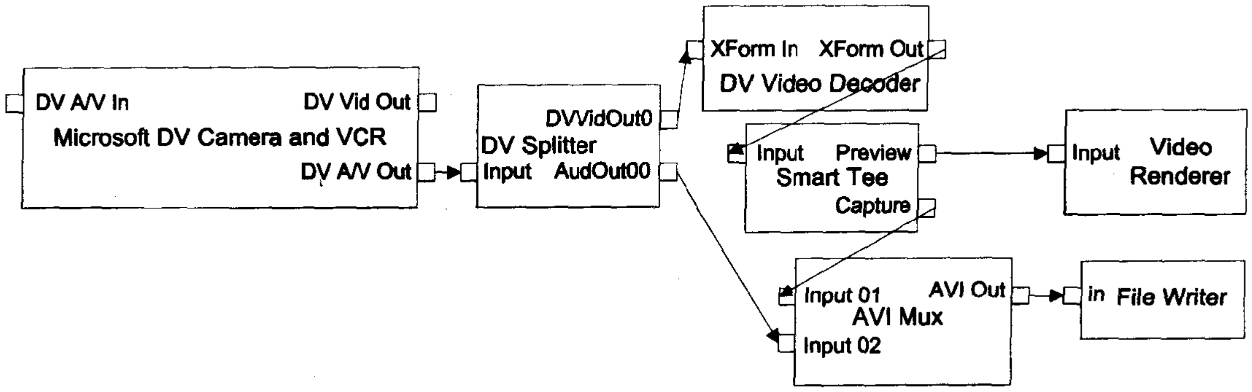


图 2.27　数码摄像机采集用的 Filter Graph

AVCap程序设计了另外一个控制类CCaptureController（该类从CGraphController类派生），专门负责采集用的Filter Graph的构建。

```
//
// CCaptureController.cpp
//
// 重写虚函数ConnectFilters，完成采集用的输入部分与输出部分的Filter连接
BOOL CCaptureController::ConnectFilters(void)
{
    BOOL pass = FALSE;
    // 首先连接视频预览支路
    IPin * pOut = mInputFilters->GetPreviewPin();
    IPin * pIn  = mVideoRenderer->GetInputPin();
    if (pOut && pIn)
    {
        pass = mGraph->ConnectFilters(pOut, pIn);
    }
    // 将视频流连到AVI Mux上，如果使用压缩，则先接入压缩Filter
    pOut = mInputFilters->GetVideoOutPin();
    if (pass && mVideoEncoder && mVideoEncoder->GetFilter())
    {
```

```
        // 接入视频压缩Filter（省略）
        // ……
    }
    pIn = mAviMux->GetUnconnectedInputPin();
    if (pass && pOut && pIn)
    {
        pass = mGraph->ConnectFilters(pOut, pIn);
    }
    // 将音频流也连到AVI Mux上
    pOut = mInputFilters->GetAudioOutPin();
    pIn  = mAviMux->GetUnconnectedInputPin();
    if (pass && pOut && pIn)
    {
        pass = mGraph->ConnectFilters(pOut, pIn);
    }
    // 将AVI Mux与File Writer相连
    pOut = mAviMux->GetOutputPin();
    pIn  = mFileWriter->GetInputPin();
    if (pass && pOut && pIn)
    {
        pass = mGraph->ConnectFilters(pOut, pIn);
    }
    return pass;
}
// 重写虚函数CreateOutputFilters，完成采集用的输出部分Filter的创建
BOOL CCaptureController::CreateOutputFilters(void)
{
    BOOL pass = CGraphController::CreateOutputFilters();
    if (pass)
    {
        // 创建AVI Mux Filter（用于音视频合成）
        mAviMux = new CAviMuxFilter(mGraph->GetGraph());
        pass = mAviMux->CreateFilter();
        // 创建File Writer Filter（用于将AVI数据保存为文件）
        mFileWriter = new CFileWriterFilter(mGraph->GetGraph());
        pass = mFileWriter->CreateFilter();
        // 给File Writer Filter设置文件路径
        mFileWriter->SetDestFile(mLiveCapture->GetOutputFile());
        // 根据实际需要，创建音视频压缩Filter（省略）
        // ……
    }
    return pass;
}
// 重写虚函数ReleaseOutputFilters：销毁输出部分的Filter
void CCaptureController::ReleaseOutputFilters(void)
{
    SAFE_DELETE(mVideoEncoder);
    SAFE_DELETE(mAviMux);
    SAFE_DELETE(mFileWriter);
    CGraphController::ReleaseOutputFilters();
}
```

当然，对于信号源自数码摄像机的情况，保存为AVI文件采用的Filter Graph不只图2.27一种，图2.28和图2.29的Filter Graph也是可以的。图2.28保存的就是所谓的Type-1类型的DV

AVI文件（DV数据直接写入AVI文件），而图2.29保存的是Type-2类型的DV AVI文件（DV的音视频数据分离之后再写成AVI文件）。在实际应用中，图2.28的Filter Graph效率更高一点。（AVCap程序并没有实现图2.28和图2.29的Filter Graph的构建，有兴趣的话读者可以当作练习去完成。）

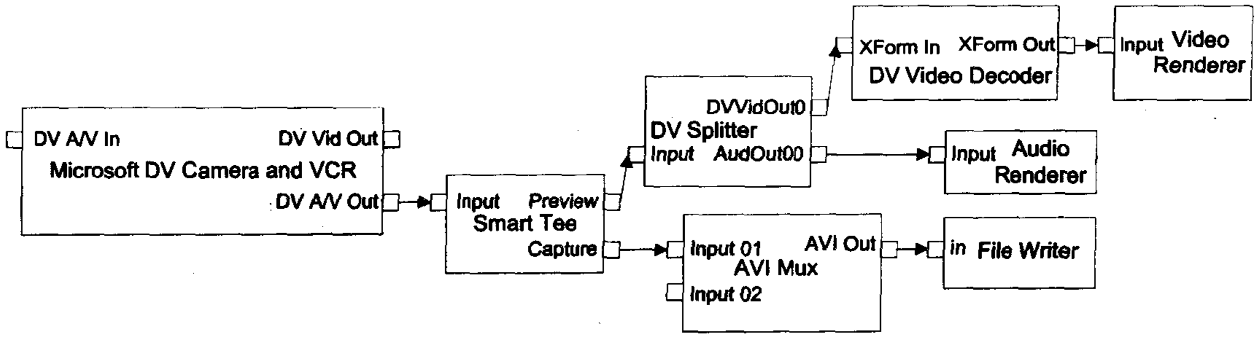


图 2.28　将 DV 数据保存为 Type-1 类型的 AVI 文件

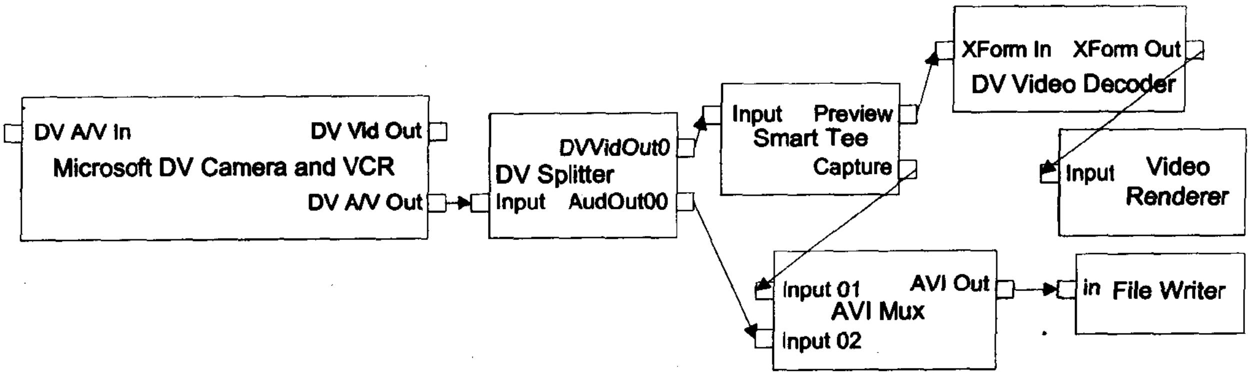


图 2.29　将 DV 数据保存为 Type-2 类型的 AVI 文件

Type-1类型和Type-2类型的DV AVI文件的播放链路如图2.30和图2.31所示。

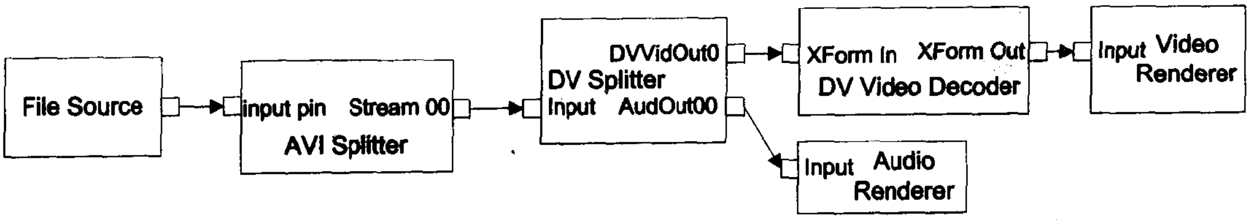


图 2.30　Type-1 类型的 DV AVI 文件的播放链路

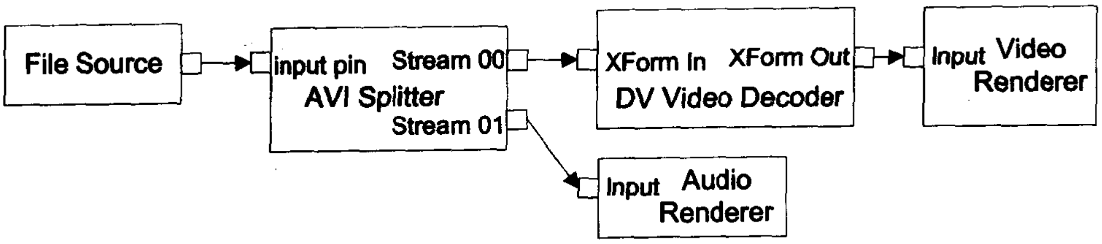


图 2.31　Type-2 类型的 DV AVI 文件的播放链路

2.2.11 视频的压缩方案

如果音视频数据都不经过压缩而直接保存为AVI文件，那么最后的文件尺寸将是很大的。相对于音频，视频的数据量更大，通常应该首先考虑对视频内容进行压缩，然后再保存到文件中去。也就是说，在将视频输出Pin连接到AVI Mux Filter之前，首先经过一个编码Filter进行视频压缩，如图2.32所示（以AGP接口的采集设备为例）。

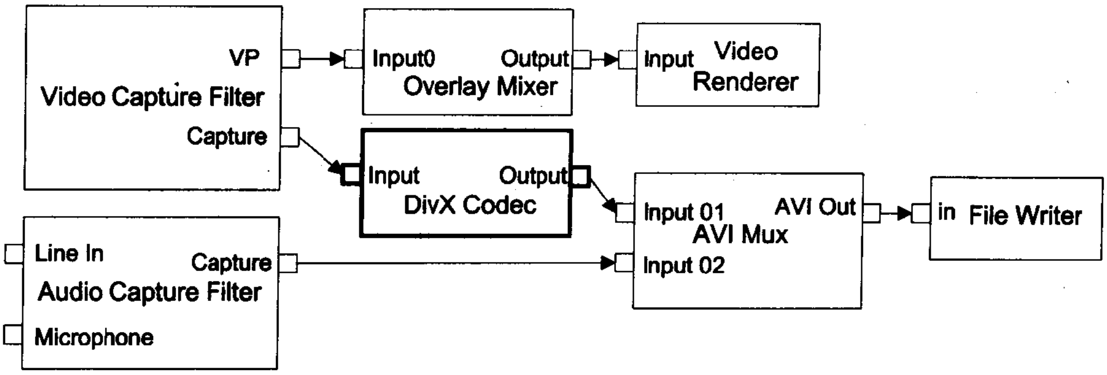


图 2.32 视频压缩后再保存为 AVI 文件的 Filter Graph

作为演示，AVCap程序使用了5.0.2版本的DivX编码器，实现如下：

```
//
// CCaptureController.cpp
//
// 重写虚函数ConnectFilters，完成采集用的输入部分与输出部分的Filter连接
BOOL CCaptureController::ConnectFilters(void)
{
    BOOL pass = FALSE;

    // 其他Filter连接（省略）
    // ……

    // 将视频流连到AVI Mux上，如果使用压缩，则先接入压缩Filter
    pOut = mInputFilters->GetVideoOutPin();
    if (pass && mVideoEncoder && mVideoEncoder->GetFilter())
    {
        // 得到视频压缩Filter的输入Pin
        pIn = mVideoEncoder->GetPin(TRUE);
        if (pOut && pIn)
        {
            // 将上一级视频输出Pin连接到压缩Filter的输入Pin上
            pass = mGraph->ConnectFilters(pOut, pIn);
        }
        // 得到视频压缩Filter的输出Pin
        pOut = mVideoEncoder->GetPin(FALSE);
    }

    // 将视频压缩Filter继续往下连接（省略）
    // ……
```

```
    return pass;
}
// 创建采集用的输出部分的Filter（必要时创建视频压缩Filter）
BOOL CCaptureController::CreateOutputFilters(void)
{
    BOOL pass = CGraphController::CreateOutputFilters();
    if (pass)
    {
        // 其他Filter创建（省略）
        // ……

        // 如果用户选择DivX压缩，则创建压缩Filter
        if (mLiveCapture->GetEncodingType() == ET_DIVX)
        {
            // 通过枚举Video Compressors目录来创建DivX压缩Filter
            IBaseFilter* pFlt = UDsUtils::CreateCompressor(TRUE,
                "DivX 5.0.2 Codec");
            if (pFlt)
            {
                // 如果创建成功，将压缩Filter加入到Filter Graph中
                mVideoEncoder = new CDXFilter(mGraph->GetGraph());
                mVideoEncoder->Attach(pFlt, "DivX Encoder");
                pFlt->Release();
                pass = TRUE;
            }
            else
            {
                // 系统中可能没有预先安装这个压缩Filter，则创建失败
                pass = FALSE;
                AfxMessageBox("DivX Encoder failed to create!");
            }
        }
    }
    return pass;
}

//
// UDsUtils.cpp
//
// 通过系统枚举创建压缩Filter
IBaseFilter * UDsUtils::CreateCompressor(BOOL inIsVideo, const char *
    inName)
{
    // "Video Compressors"目录——CLSID_VideoCompressorCategory
    // "Audio Compressors"目录——CLSID_AudioCompressorCategory
    GUID guid = inIsVideo ?
        CLSID_VideoCompressorCategory:CLSID_AudioCompressorCategory;
    return CreateHardwareFilter(guid, inName);
}
```

提示：本书配套光盘提供了5.0.2版本的DivX编解码器的安装程序——DivX502 Bundle.exe。在进行DivX压缩的采集之前，请确认该编码器已经安装。另外，如果你的系统中安装的是不同版本的DivX编码器，也可以通过修改上述程序中的UDsUtils::CreateCompressor函数调用的参数（指定为正确的Filter名字即可）来支持使用不同版本的编码器。

当然，视频采集也可以不保存为AVI文件，如保存为MPEG文件，此时需要用到MPEG的音视频编码Filter、音视频合成Filter等。DirectShow本身并不提供这些Filter，因此我们必须自行开发。另外，因为这些Filter要用在实时采集的场合，通常要求它们具有较高的性能。如果它们达不到实时处理的要求，采集时产生丢帧现象将是必然的。采集保存为MPEG文件的Filter Graph可以参考图2.33。

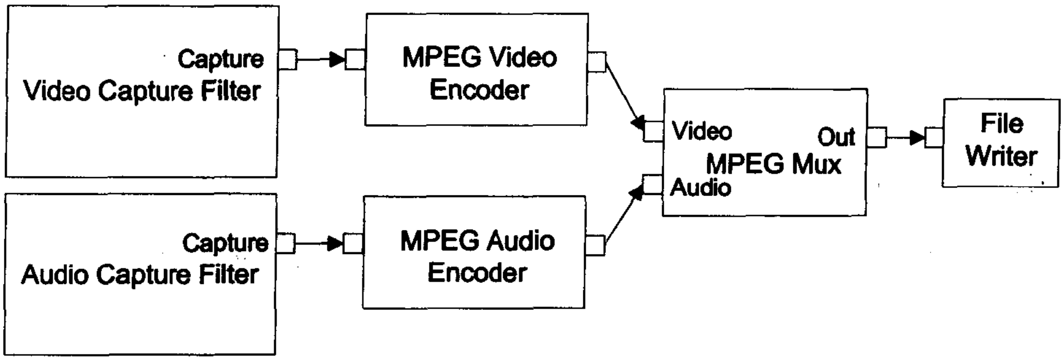


图 2.33　采集保存为 MPEG 文件的 Filter Graph

另外，还有一些采集设备支持硬件压缩，能够直接输出压缩的音视频流。这种采集设备的Filter Graph如图2.34所示。

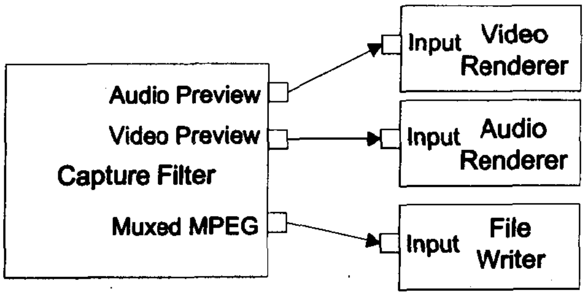


图 2.34　使用硬件压缩采集设备的 Filter Graph

提示：用硬件实现的音视频压缩比起软件实现来，效率通常要高很多。因此在视频监控应用领域中，大都采用了带硬件压缩的采集卡。这种采集卡大都提供了二次开发包，本地采集时未必需要构建如图2.34所示的Filter Graph。但是，如果需要扩展网络监控功能，本地端就必须将音视频数据（通常是压缩过的）发送至网络客户端。在客户端接收、解码、播放音视频内容，就可以应用DirectShow技术了。关于网络部分的应用，本书将在第3章作详细介绍。

2.3 电视接收

2.3.1 预备工作

有些采集卡带有（模拟）电视接收的功能，AVCap程序也对电视接收提供了支持。因为电视卡是采集卡的一种特例，所以AVCap程序从CAnalogInputFilters类派生了一个子类CTVInputFilters，专门负责采集卡基本操作之外的与电视相关的操作。

电视卡的Filter Graph通常构建为如图2.10所示的形式。真正的电视接收功能是由TV Tuner Filter实现的。这个Filter上主要实现了IAMTVTuner接口（IAMTVTuner接口从IAMTuner接口继承），应用程序通过这个接口可以实现电视频道的自动搜索、设定等一系列操作。CTVInputFilters类对IAMTVTuner接口常用的一些函数进行了包装。

```
//
// CTVInputFilters.cpp
//
// 创建电视卡输入部分的Filter
BOOL CTVInputFilters::Create(void)
{
    // 构建普通采集卡应用的Filter Graph
    BOOL pass = CAnalogInputFilters::Create();
    if (pass)
    {
        // 获得电视操作相关的接口
        FindTVTuner();
    }
    return pass;
}
// IAMTVTuner *   mTvTuner;
// 查询得到IAMTVTuner接口（该接口实际在TV Tuner Filter上实现）
BOOL CTVInputFilters::FindTVTuner(void)
{
    if (mTvTuner)
    {
        return TRUE;
    }
    if (mGraphBuilder)
    {
        mGraphBuilder->FindInterface(&LOOK_UPSTREAM_ONLY,
            &MEDIATYPE_Video, mVideoCapture->GetFilter(),
            IID_IAMTVTuner, (void **)&mTvTuner);
    }
    return (mTvTuner != NULL);
}
// 得到TV Tuner上已经设置的国家代码（即接收指定国家的电视信号）
// 中国：86  美国：1  日本：81  韩国：82  英国：44  法国：33  德国：49
long CTVInputFilters::GetCountryCode(void)
{
```

```
    long  code = -1;
    if (mTvTuner)
    {
        mTvTuner->get_CountryCode(&code);
    }
    return code;
}
// 设置欲接收电视所属的国家代码
BOOL CTVInputFilters::SetCountryCode(long inCode)
{
    HRESULT hr = E_FAIL;
    if (mTvTuner)
    {
        hr = mTvTuner->put_CountryCode(inCode);
        if (SUCCEEDED(hr))
        {
            // 同时设置相应的调谐空间
            mTvTuner->put_TuningSpace(inCode);
        }
    }
    return SUCCEEDED(hr);
}
// 得到TV Tuner上已经设置的信号输入方式：无线电视或有线电缆
TunerInputType CTVInputFilters::GetTunerInputType(void)
{
    TunerInputType type = TunerInputAntenna;
    if (mTvTuner)
    {
        long  inputConnect = 0;
        // 首先得到当前连接的输入端子
        if (SUCCEEDED(mTvTuner->get_ConnectInput(&inputConnect)))
        {
            // 然后得到当前输入端子上的信号输入方式
            mTvTuner->get_InputType(inputConnect, &type);
        }
    }
    return type;
}
// 设置信号输入方式：无线电视或有线电缆
BOOL CTVInputFilters::SetTunerInputType(TunerInputType inType)
{
    HRESULT hr = E_FAIL;
    if (mTvTuner)
    {
        long  inputConnect = 0;
        // 首先得到当前连接的输入端子
        hr = mTvTuner->get_ConnectInput(&inputConnect);
        if (SUCCEEDED(hr))
        {
            // 然后给当前输入端子设置信号输入方式
            hr = mTvTuner->put_InputType(inputConnect, inType);
        }
```

```
    }
    return SUCCEEDED(hr);
}
// 得到当前设置的频道号
long CTVInputFilters::GetTVChannel(void)
{
    if (mTvTuner)
    {
        long freqIndex;
        long videoSub, audioSub;
        if (SUCCEEDED(mTvTuner->get_Channel(&freqIndex, &videoSub,
            &audioSub)))
        {
            return freqIndex;
        }
    }
    return 0;
}
// 设置频道号
BOOL CTVInputFilters::SetTVChannel(long inFrequencyIndex)
{
    if (mTvTuner)
    {
        long signalStrength  = AMTUNER_NOSIGNAL;
        long videoSubChannel = AMTUNER_SUBCHAN_DEFAULT;
        long audioSubChannel = AMTUNER_SUBCHAN_DEFAULT;
        mTvTuner->put_Channel(inFrequencyIndex, videoSubChannel,
            audioSubChannel);
        // 检测该频道的信号是否稳定
        mTvTuner->SignalPresent(&signalStrength);
        if (signalStrength != AMTUNER_NOSIGNAL)
        {
            return TRUE;
        }
    }
    return FALSE;
}
// 得到指定国家、指定电视接收方式的频道号的范围
BOOL CTVInputFilters::GetChannelMinMax(long * outMin, long * outMax)
{
    if (mTvTuner)
    {
        if (SUCCEEDED(mTvTuner->ChannelMinMax(outMin, outMax)))
        {
            return TRUE;
        }
    }
    return FALSE;
}
// 得到TV Tuner的工作（视频信号）制式
Signal_Resolution CTVInputFilters::GetTunerResolution(void)
{
```

```
    if (mTvTuner)
    {
        long resolution = 0;
        mTvTuner->get_TVFormat(&resolution);
        if (cNTSCMaskBits & resolution)
        {
            return SR_NTSC;
        }
        else if (cPALMaskBits & resolution)
        {
            return SR_PAL;
        }
        else if (cSECAMMaskBits & resolution)
        {
            return SR_SECAM;
        }
    }
    return SR_UNKNOWN;
}
```

提示：实际应用中，一个电视卡只能接收某一种制式的电视信号。TV Tuner的工作制式通常情况下是无法通过应用程序去改变的。应用程序能做的，就是从TV Tuner上获取它的工作制式，然后将这个制式设置给采集设备Filter，以便采集设备能对电视输入信号进行正确的解码。

2.3.2 电视选台

如果用户是第一次使用电视接收功能，一般首先要进行一次全频道搜索。图2.4是TV Tuner Filter的属性页，单击界面上的AutoTune按钮就可以启动一次全频道的自动搜索过程。这个过程中所做的工作，AVCap程序给出了如下演示：

```
//
// CTVInputFilters.cpp
//
// 演示Auto Tune过程
void CTVInputFilters::DemoTVChannelAutoTune(void)
{
    if (mTvTuner)
    {
        long minChannel = 0, maxChannel = 0;
        long foundSignal = AMTUNER_NOSIGNAL;
        // 得到频道号的范围
        HRESULT hr = mTvTuner->ChannelMinMax(&minChannel, &maxChannel);
        for (long i = minChannel; i <= maxChannel; i++)
        {
            // 对当前频道号进行调谐
            // 驱动程序中可以查询得到频道号对应的电视信号实际的载频
            hr = mTvTuner->AutoTune(i, &foundSignal);
            // 检查当前频道信号是否稳定
            // 假设AMTUNER_HASNOSIGNALSTRENGTH也表示该频道信号正常
```

```
            if (foundSignal != AMTUNER_NOSIGNAL)
            {
                // 搜到了一个有信号的频道
                // 可以将该频道号进行保存以方便用户使用（暂未实现）
                // ……
            }
            // 保存各频道完成调谐的信息
            if (SUCCEEDED(hr))
            {
                mTvTuner->StoreAutoTune();
            }
        }
    }
```

提示：*在启动频道的自动搜索之前，请确认已经正确设置了接收电视所属的国家代码、调谐空间（Tuning Space）、信号输入方式等参数。*

经过全频道搜索之后，用户就可以选择自己感兴趣的频道观看电视节目了。通过图2.4的界面，在Channel中输入频道号，然后按Enter键即可。不过，图2.4的界面使用起来并不方便：用户不知道哪些频道有节目、哪些频道没有节目；不能给频道命名（数字形式的频道号很难记住，现在已经很少用了）等。一般在商业软件中，都会自行实现一个电视的操作界面。在这个界面中，可以调用IAMTVTuner的各个接口函数进行控制，完全可以实现图2.4的界面中类似的功能。

提示：*AVCap程序并没有定制自己的电视操作界面，有兴趣的读者可以把它当作练习来完成。*

2.3.3 支持VBI

众所周知，电视图像的摄取及显示是以电子束对图像扫描的方式进行的。电子束完成一行扫描之后回到另一行起点的这段逆程时间称为行回扫期或行消隐期。电子束完成一帧扫描之后回到另一帧起点的这段逆程时间称为帧回扫期或帧消隐期。我国的广播电视采用的是625行隔行扫描方式。隔行扫描是将一帧图像分两次（场）扫描。第一场先扫出1、3、5、7等奇数行光栅，第二场扫2、4、6、8等偶数行光栅。通常将扫奇数行的场叫奇场，扫偶数行的场叫偶场。这两场之间的回扫时间称为场回归期或场消隐期。消隐期的长度为25行，除去场同步的槽、齿脉冲外，奇场消隐期中的第7~22行、偶场消隐期中的第320~325行均可插入其他信号。（图文电视便是将各类图形、文字信息以数字信号的形式叠加在广播电视信号场消隐期的若干行上，然后与正常广播电视信号一起发送的数据广播系统。）

VBI（Vertical Blanking Interval，垂直消隐间隔），也就是上述的场回归期（或场消隐期）。如果电视信号中夹带了VBI信息，DirectShow同样支持把它们与电视图像一起显示出来。

常见的VBI信号有两种：CC（Closed Caption）和WST（World Standard Teletext）。CC信号的显示可以构建如图2.35所示的Filter Graph，WST信号的显示可以构建如图2.36所示的Filter Graph。

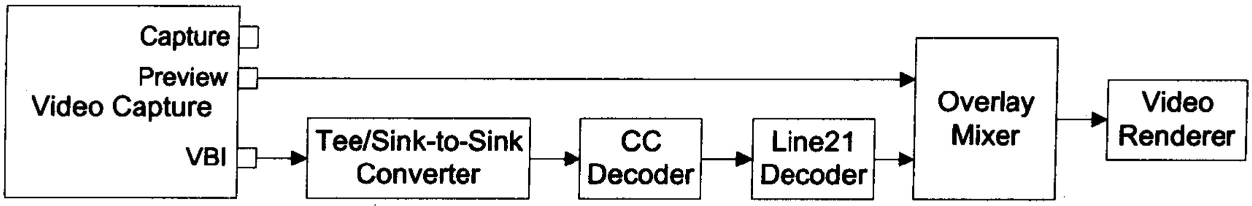


图 2.35　显示 CC 信号的 Filter Graph

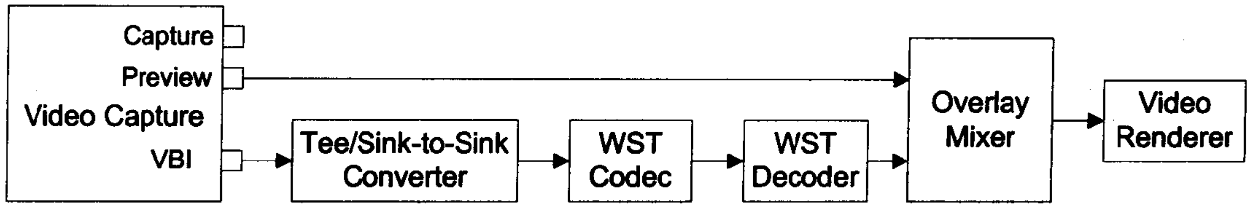


图 2.36　显示 WST 信号的 Filter Graph

需要注意的是，有的采集设备Filter可能还含有VPVBI输出Pin。此时，必须为这个Pin接上一个VBI Surface Allocator Filter，构建的Filter Graph如图2.37所示（以显示CC信号为例）：

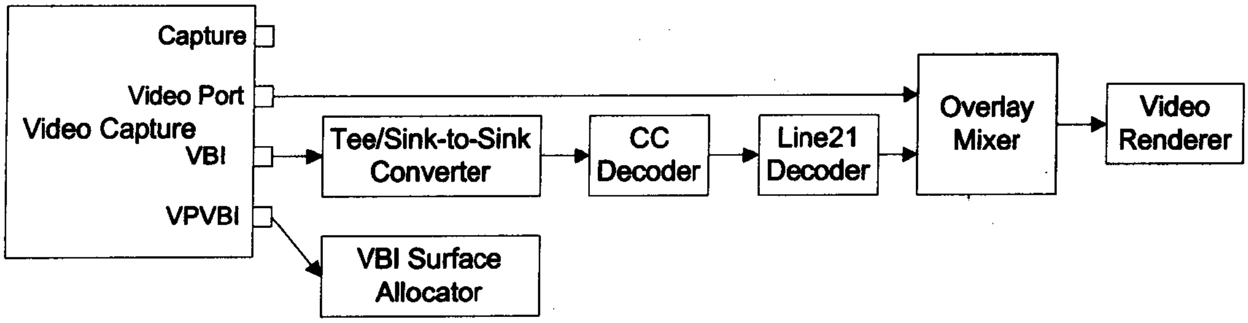


图 2.37　带有 VPVBI 输出 Pin 的采集设备显示 CC 信号的 Filter Graph

图2.35的Filter Graph很容易构建，因为可以借助于ICaptureGraphBuilder2接口的RenderStream函数。

```
//
// CTVInputFilters.cpp
//
// 构建解码CC信号的Filter链路
BOOL CTVInputFilters::TryCCFilters(void)
{
    if (!mGraphBuilder)
    {
        return FALSE;
    }

    // 为CC类型的Pin构建Filter链路
    // 如果采集设备有VPVBI输出Pin，则同时为它构建Filter链路（自动）
    HRESULT hr = mGraphBuilder->RenderStream(&PIN_CATEGORY_CC, 0,
        mVideoCapture->GetFilter(), 0, 0);
    if (hr != NOERROR)
```

```
    {
        // 如果CC类型的Pin的Filter链路构建不成功，则指定VBI类型再试
        hr = mGraphBuilder->RenderStream(&PIN_CATEGORY_VBI, 0,
            mVideoCapture->GetFilter(), 0, 0);
    }
    return SUCCEEDED(hr);
}
```

提示：在AVCap程序的设计中，当调用ICaptureGraphBuilder2::RenderStream函数构建了采集设备预览用的Filter Graph之后，会随即将采集设备Filter后面的Filter链路删除（参见CAnalogInputFilters::Create函数的实现）。此时再调用上述子类CTVInputFilters的TryCCFilters函数进行CC信号解码链路的构建时会有些问题！解决的办法一是重新设计CAnalogInputFilters类与CTVInputFilters类之间的接口函数调用时机，使得TryCCFilters函数在CAnalogInputFilters类的Filter链路删除之前被调用；二是像构建WST信号的解码链路一样，显式地加入必要的Filter，然后一个一个依次连接。

图2.36的Filter Graph构建比较麻烦，因为ICaptureGraphBuilder2接口不直接支持WST信号的解码链路的构建。

```
//
// CTVInputFilters.cpp
//
// 构建解码WST信号的Filter链路
BOOL CTVInputFilters::TryWSTFilters(void)
{
    if (!mGraphBuilder)
    {
        return FALSE;
    }

    IBaseFilter * pKernelTee = NULL;
    IBaseFilter * pWstCodec  = NULL;
    HRESULT hr = NOERROR;
    BOOL  pass = FALSE;
    // 创建一个硬件Tee Filter
    pKernelTee = UDsUtils::CreateHardwareFilter(AM_KSCATEGORY_SPLITTER,
        "Tee/Sink-to-Sink Converter");
    pass = (pKernelTee != 0);
    if (pass)
    {
        // 将硬件Tee Filter加入Filter Graph
        mGraph->GetGraph()->AddFilter(pKernelTee, L"Kernel Tee");
        // 创建WST Codec Filter
        pWstCodec  = UDsUtils::CreateHardwareFilter(
            AM_KSCATEGORY_VBICODEC, "WST Codec");
        pass = (pWstCodec != 0);
    }
    if (pass)
    {
```

```
        // 将WST Codec加入到Filter Graph
        mGraph->GetGraph()->AddFilter(pWstCodec, L"WST Codec");
        // 连接采集设备Filter上的VBI输出Pin
        hr = mGraphBuilder->RenderStream(&PIN_CATEGORY_VBI, 0,
            mVideoCapture->GetFilter(), pKernelTee, pWstCodec);
        pass = SUCCEEDED(hr);
    }
    if (pass)
    {
        // 连接WST Codec后面的Filter链路
        hr = mGraphBuilder->RenderStream(0, 0, pWstCodec, 0, 0);
        pass = SUCCEEDED(hr);
    }
    if (pass)
    {
        // 检查VPVBI输出Pin的连接状态
        Verify_VPVBI_Connection();
    }
    // 释放使用过的接口
    if (pKernelTee)
    {
        pKernelTee->Release();
        pKernelTee = 0;
    }
    if (pWstCodec)
    {
        pWstCodec->Release();
        pWstCodec = 0;
    }
    return pass;
}
// 如果采集设备Filter含有一个VPVBI输出Pin，则为它连上VBI Surface Allocator
void CTVInputFilters::Verify_VPVBI_Connection(void)
{
    if (mGraphBuilder)
    {
        IPin * pVPVBI = NULL;
        // 查看采集设备Filter上是否有VPVBI输出Pin
        HRESULT hr = mGraphBuilder->FindPin(
            mVideoCapture->GetFilter(),
            PINDIR_OUTPUT,              // Pin的方向
            &PIN_CATEGORY_VIDEOPORT_VBI, // Pin的类型
            NULL,
            TRUE,
            0,
            &pVPVBI
            );
        if (hr == S_OK)
        {
            // 创建一个VBI Surface Allocator，并把它加入Filter Graph
            IBaseFilter * pSurf = NULL;
            hr = CoCreateInstance(CLSID_VBISurfaces, NULL,
```

```
            CLSCTX_INPROC_SERVER, IID_IBaseFilter, (void **)&pSurf);
        if (SUCCEEDED(hr))
        {
            hr = mGraph->GetGraph()->AddFilter(pSurf, L"VBI Surf");
            // 将VPVBI输出Pin连接到VBI Surface Allocator上
            hr = mGraphBuilder->RenderStream(
                NULL,NULL,pVPVBI,0,pSurf);
            pSurf->Release();
        }
        pVPVBI->Release();
      }
    }
}
```

当然，如果成功构建了CC信号（或WST信号）的解码链路，在进行视频预览的时候仍然可以打开和关闭这些信息的显示。AVCap程序给出了如下的演示代码：

```
//
// CTVInputFilters.cpp
//
#include <Il21dec.h>
#include <iwstdec.h>
// 打开或关闭CC信息的显示
BOOL CTVInputFilters::DisplayClosedCaption(BOOL inIsOn)
{
    if (!mGraphBuilder)
    {
        return FALSE;
    }

    // 获得IAMLine21Decoder接口（在Line21 Decoder Filter上实现）
    IAMLine21Decoder *pLine21 = NULL;
    HRESULT hr = mGraphBuilder->FindInterface(
        &LOOK_DOWNSTREAM_ONLY,
        NULL,
        mVideoCapture->GetFilter(),
        IID_IAMLine21Decoder,
        (void**)&pLine21
        );
    if (SUCCEEDED(hr))
    {
        pLine21->SetServiceState(inIsOn ? AM_L21_CCSTATE_On :
            AM_L21_CCSTATE_Off);
        pLine21->Release();
    }
    return SUCCEEDED(hr);
```

```
}
// 打开或关闭WST信息的显示
BOOL CTVInputFilters::DisplayWSTText(BOOL inIsOn)
{
    if (!mGraphBuilder)
    {
        return FALSE;
    }

    // 获得IAMWstDecoder接口（在WST Decoder Filter上实现）
    IAMWstDecoder *pwst = NULL;
    HRESULT hr = mGraphBuilder->FindInterface(
        &LOOK_DOWNSTREAM_ONLY,
        NULL,
        mVideoCapture->GetFilter(),
        IID_IAMWstDecoder,
        (void**)&pwst
        );
    if (SUCCEEDED(hr))
    {
        pwst->SetServiceState(inIsOn ? AM_WST_STATE_On :
            AM_WST_STATE_Off);
        pwst->Release();
    }
    return SUCCEEDED(hr);
}
```

2.4 程序导读

2.4.1 一种不错的设计模式

定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新，这就是观察者（Observer）模式——一种很实用的设计模式！

很多程序在设计的时候将界面表示与底下的应用数据分离，这样，界面类和数据类就可以各自独立地复用。但是，在多个界面一起工作的时候，如何维持应用数据在各个界面上表示的一致性呢？如图2.38所示，一个表格、一个柱状图、一个饼形图，它们都依赖于同一个数据对象，但又都不知道其他界面的存在。那么，当数据在表格中改变时，如何同步更新柱状图和饼形图上的表示呢？运用观察者模式就可以很好地解决这个问题：表格在修改完数据之后能够马上通知其他依赖于该数据的界面进行同步更新，就好像表格知道有一个柱状图和一个饼形图存在一样。

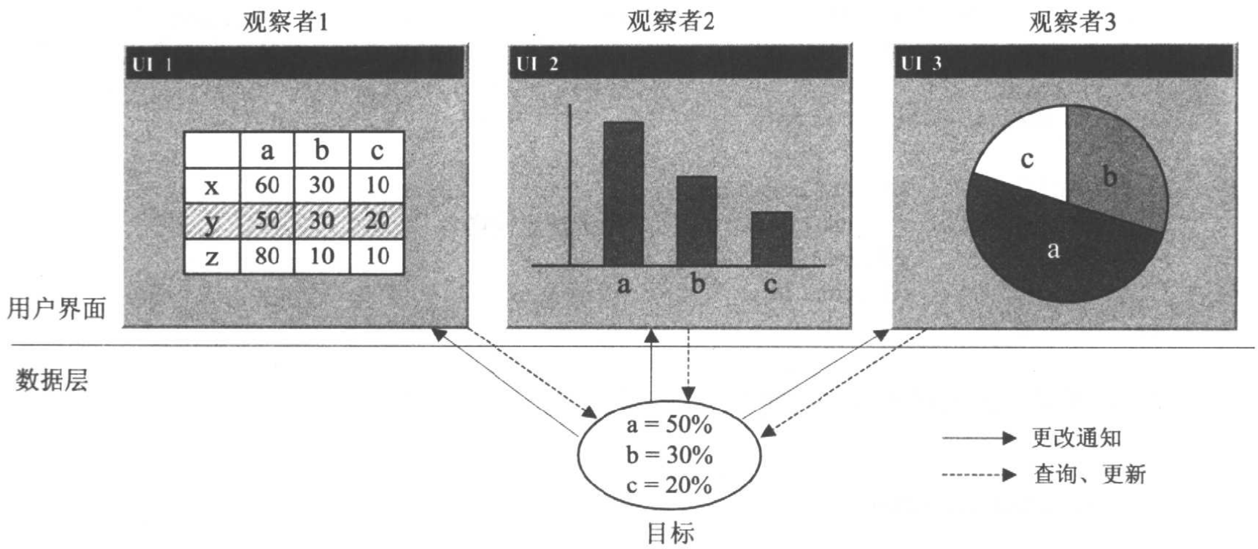


图 2.38　多界面依赖于同一个数据对象

AVCap程序在实现时也用到了这个模式。首先要设计目标类CMsgStation（就好像是一个消息发送站）和观察者类CMsgReceiver（就好像是一个消息接收器）。

```
//
// CMsgStation.h
//

#ifndef __H_CMsgStation__
#define __H_CMsgStation__

#include "CObjectList.h"

typedef long MessageT;  // 消息类型
typedef long NotifyT;   // 通知类型

class CMsgReceiver;
class CMsgStation
{
protected:
    CObjectList mReceivers;  // 维持一个所有观察者的列表
public:
    CMsgStation(void);
    virtual ~CMsgStation(void);
    // 增加一个观察者（当某个界面类希望获得数据改变的通知）
    void AddMsgReceiver(CMsgReceiver * inReceiver);
    // 删除一个观察者（当某个界面类不再需要获得数据改变的通知）
    void RemoveMsgReceiver(CMsgReceiver * inReceiver);
    // 将数据的改变通知列表中所有的观察者
    bool Broadcast(MessageT inMessage, void * ioParam = 0,
        void * ioParam2 = 0);
    bool Notify(NotifyT inNotification, long inParam1 = 0,
        long inParam2 = 0);
};
```

```
const MessageT msg_StationDestroyed = 0;  // 自定义消息：目标对象销毁了
const MessageT msg_Notify           = 1;  // 自定义消息：这是一个参数更改通知

struct SNotificationStruct
{
    NotifyT      mNotification;
    CMsgStation *  mStation;  // 指示消息发送者
    long         mParam1;
    long         mParam2;
};

#endif // __H_CMsgStation__

//
// CMsgStation.cpp
//

#include "stdafx.h"
#include "CMsgStation.h"
#include "CMsgReceiver.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
//////////////////////////////////////////////////////////////////////
CMsgStation::CMsgStation(void)
{
}
CMsgStation::~CMsgStation(void)
{
    // 发送给所有的观察者：目标对象销毁了
    Broadcast(msg_StationDestroyed, (void *)this);
}
// 增加一个观察者到列表中保存（观察者必须是CMsgReceiver的子类实例）
void CMsgStation::AddMsgReceiver(CMsgReceiver * inReceiver)
{
    if (inReceiver)
    {
        if (mReceivers.GetIndex(inReceiver) == -1)
        {
            mReceivers.Add(inReceiver);
            // 观察者同时也保存本目标对象的指针
            inReceiver->AddMsgStation(this);
        }
    }
}
// 从列表中删除一个观察者（观察者必须是CMsgReceiver的子类实例）
void CMsgStation::RemoveMsgReceiver(CMsgReceiver * inReceiver)
{
    if (inReceiver)
```

```
    {
        // 观察者同时也删除本目标对象的指针
        inReceiver->RemoveMsgStation(this);
        mReceivers.Remove(inReceiver);
    }
}
// 将数据的改变通知列表中的所有观察者
bool CMsgStation::Broadcast(MessageT inMessage, void * ioParam, void *
    ioParam2)
{
    for (int i = mReceivers.GetSize() - 1; i >= 0; i --)
    {
        CMsgReceiver * receiver = (CMsgReceiver *)mReceivers.GetAt(i);
        if (receiver)
        {
            // 实际上就是调用观察者的ReceiveMessage函数
            receiver->ReceiveMessage(inMessage, ioParam, ioParam2);
        }
    }
    return true;
}
// 另一种形式的通知函数：通知参数中携带了通知的发送者
bool CMsgStation::Notify(NotifyT inNotification, long inParam1, long
    inParam2)
{
    SNotificationStruct  notification;
    notification.mNotification = inNotification;
    notification.mStation = this;
    notification.mParam1  = inParam1;
    notification.mParam2  = inParam2;
    return Broadcast(msg_Notify, &notification);
}

//
// CMsgReceiver.h
//

#ifndef __H_CMsgReceiver__
#define __H_CMsgReceiver__

#include "CMsgStation.h"

class CMsgReceiver
{
protected:
    CObjectList mStations;  // 维持一个所有目标的列表
public:
    CMsgReceiver(void);
    virtual ~CMsgReceiver(void);
    // 维护目标列表
    void AddMsgStation(CMsgStation * inStation);
    void RemoveMsgStation(CMsgStation * inStation);
```

```
    // 目标对象发出通知的响应函数
    // 注意：响应函数都是虚函数，可以供子类重写以响应不同的通知类型
    virtual bool ReceiveMessage(MessageT inMessage, void * ioParam,
        void * ioParam2);
    virtual bool Respond(NotifyT inNotification, CMsgStation * inStation,
        long inParam1, long inParam2);
};

#endif // __H_CMsgReceiver__

//
// CMsgReceiver.h
//

#include "stdafx.h"
#include "CMsgReceiver.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
//////////////////////////////////////////////////////////////////////
CMsgReceiver::CMsgReceiver(void)
{
}
CMsgReceiver::~CMsgReceiver(void)
{
    // 当观察者销毁时，将自己从所有的目标对象列表中删除
    for (int i = mStations.GetSize() - 1; i >= 0; i --)
    {
        CMsgStation * station = (CMsgStation *) mStations.GetAt(i);
        if (station)
        {
            station->RemoveMsgReceiver(this);
        }
    }
}
// 增加一个目标对象到列表中保存（目标对象必须是CMsgStation的子类实例）
void CMsgReceiver::AddMsgStation(CMsgStation * inStation)
{
    mStations.Add(inStation);
}
// 从列表中删除一个目标对象（目标对象必须是CMsgStation的子类实例）
void CMsgReceiver::RemoveMsgStation(CMsgStation * inStation)
{
    mStations.Remove(inStation);
}
// 目标对象发出通知的响应函数：与CMsgStation::Broadcast对应
bool CMsgReceiver::ReceiveMessage(MessageT inMessage, void * ioParam, void
    * ioParam2)
{
```

```
    switch (inMessage)
    {
    case msg_Notify:
        {
            SNotificationStruct * notification = (SNotificationStruct
                *)ioParam;
            return Respond(notification->mNotification,
                notification->mStation,notification->mParam1,
                notification->mParam2);
        }

    case msg_StationDestroyed:
        RemoveMsgStation((CMsgStation *)ioParam);
        return true;
    }
    return false;
}
// 另一个通知响应函数：与CMsgStation::Notify对应
bool CMsgReceiver::Respond(NotifyT inNotification, CMsgStation * inStation,
    long inParam1, long inParam2)
{
    return false;
}
```

然后将目标类和观察者类分别作为数据类和界面类的父类。在界面类中调用数据对象上的CMsgStation::AddMsgReceiver(this)，表示当前界面是指定数据对象的观察者；并在界面类中重写虚函数CMsgReceiver::ReceiveMessage或CMsgReceiver::Respond，用于对数据对象发出的通知作出响应。以AVCap程序中的实现为例——当系统中安装有多个视频采集设备时，通过主界面上的组合框选择其中一个作为当前工作的设备。此时需要更新数据对象中的参数，随后重构Filter Graph（由数据对象发出的通知驱动）。

```
//
// AVCapDlg.h
//
class CAVCapDlg:public CDialog,public CMsgReceiver//从CMsgReceiver类继承
{
protected:
    CLiveCapture    mLiveCapture;     // 数据对象
    // 其他成员定义（省略）
    // ……
};

//
// CLiveCapture.h
//
class CLiveCapture : public CMsgStation   // 从CMsgStation类继承
{
private:
    CString     mVideoDevice;     // 视频采集设备名字
    // 其他成员定义（省略）
    // ……
```

```
public:
    void SetVideoDevice(const char * inDevice);
};

//
// AVCapDlg.cpp
//
CAVCapDlg::CAVCapDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CAVCapDlg::IDD, pParent)
{
    // CAVCapDlg是CLiveCapture的观察者，
    // CLiveCapture发送消息，CAVCapDlg接收并处理消息
    mLiveCapture.AddMsgReceiver(this);
}
// 组合框中的选项改变后的响应函数
void CAVCapDlg::OnSelchangeComboDeviceVideo()
{
    // 获取当前选项的索引
    int index = mComboVideoDevices.GetCurSel();
    if (index != CB_ERR)
    {
        // 获得当前选中的设备名字
        CString deviceName;
        mComboVideoDevices.GetLBText(index, deviceName);
        // 将新的设备名字设置给参数对象
        mLiveCapture.SetVideoDevice(deviceName);
    }
}

//
// CLiveCapture.cpp
//
// const long cVideoSourceChanged   = 'lcap' + 1;
// 设置新的视频设备名字
void CLiveCapture::SetVideoDevice(const char * inDevice)
{
    // 当当前工作的视频采集设备改变时，保存这个新设备，并发出通知
    if (mVideoDevice.Compare(inDevice) != 0)
    {
        mVideoDevice = inDevice;
        // 检测新设备的类型
        DecideDeviceType(inDevice);
        if (mNotifying)
        {
            // 向所有对视频采集设备改变感兴趣的观察者发送通知
            Notify(cVideoSourceChanged);
        }
    }
}

//
// AVCapDlg.cpp
```

```
//
// 响应数据对象发出的通知
bool CAVCapDlg::Respond(NotifyT inNotification, CMsgStation * inStation,
    long inParam1, long inParam2)
{
    switch (inNotification)
    {
    case cVideoSourceChanged:   // 当前工作视频采集设备改变了
        // 重构Filter Graph
        CreateController();
        return true;
    }
    return CMsgReceiver::Respond(inNotification, inStation, inParam1,
        inParam2);
}
```

提示：在AVCap程序中虽然没有多个界面对数据的改变感兴趣，但是如果以后要朝这个方向扩展就会很方便。另外，像这样一种数据驱动的设计模式，还是很值得推荐的。

观察者模式不仅能解决对象间的一对多依赖问题，还能解决复合对象的双向协作问题。以AVCap程序为例，控制类CGraphController对象包含CDXGraph类的实例（它们是一种对象间的复合关系），CGraphController对象到CDXGraph实例的协作是从上往下的，在控制类中可以通过调用CDXGraph实例的各个公共函数来实现，但在CDXGraph类中如何从下往上使用CGraphController对象的功能呢？（如Filter Graph运行时出错了，需要让CGraphController对象作出相应的处理）有一种方法是，CDXGraph类中保存一个CGraphController对象的指针。如此设计，目的是达到了，但同时也将导致CDXGraph和CGraphController两个类产生紧密耦合关系，不利于CDXGraph类的复用。但是，如果运用观察者模式（CDXGraph对象是目标，CGraphController对象是观察者），当Filter Graph运行时出错了，可以调用CMsgStation::Broadcast(msg_FilterGraphError)发出一个自定义消息，这时只需在控制类中对msg_FilterGraphError消息作出相应的处理，也就等于实现了CDXGraph实例到CGraphController对象从下往上的协作。这样的设计，目标和观察者之间的耦合是抽象的和最小的。

提示：在本书后续章节的几个实例程序中，CMsgStation和CMsgReceiver两个类还会被多次用到，因此，在这里了解这两个类的实现原理以及使用方法是很有必要的。关于观察者模式更详细的介绍，请读者参考《设计模式——可复用面向对象软件的基础》一书的5.7节。

2.4.2 程序结构

首先来了解一下AVCap项目的VC工作区结构，如图2.39所示。

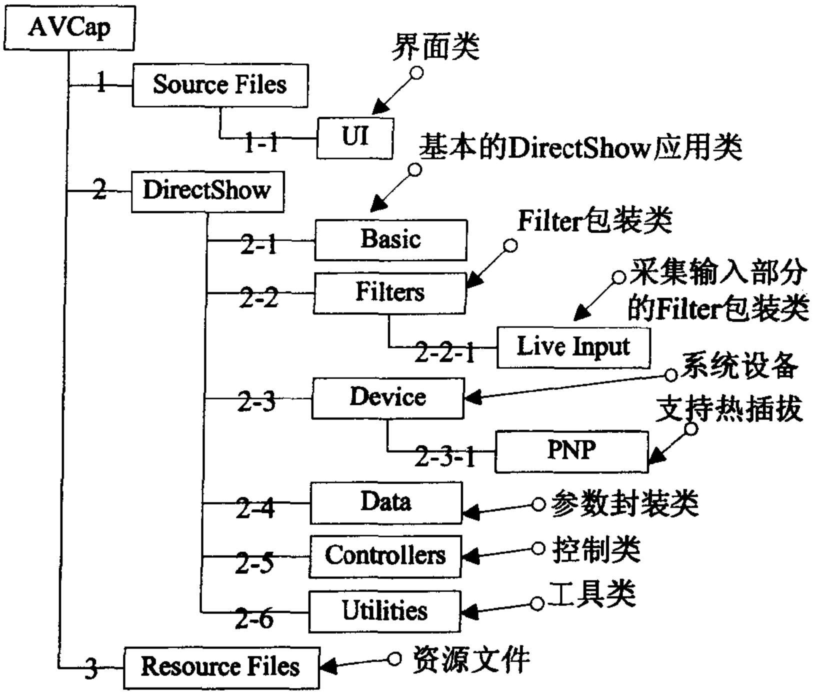


图 2.39　AVCap 项目的 VC 工作区结构

AVCap程序定义了很多类，这些类的清单如表2.4所示。

表2.4　AVCap程序中的所有类清单

类名	所在目录	功能描述
CAVCapDlg	1-1	主界面
CAudioSourcePage	1-1	音频采集设备的设置界面
COutputPage	1-1	采集输出参数的设置界面
CVideoSourcePage	1-1	视频采集设备的设置界面
CGraphController	2-5	控制类的基类，用于定义控制接口、以及Filter Graph的构建
CPreviewController	2-5	负责预览用的Filter Graph的构建
CCaptureController	2-5	负责采集用的Filter Graph的构建
CLiveCapture	2-4	保存各种参数（包括音视频采集设备的名字、设备类型、采集输出图像的尺寸、视频的压缩方法、当前的工作模式等）
CLiveInputFilters	2-2-1	输入部分Filter的包装类的基类，用于定义各个公共接口
CAnalogInputFilters	2-2-1	负责普通采集设备输入的包装类
CTVInputFilters	2-2-1	支持带电视接收功能的采集设备输入的包装类
CDVInputFilters	2-2-1	负责数码摄像机输入的包装类
CVideoDevices	2-3	负责搜集、保存系统中所有的视频采集设备信息
CAudioDevices	2-3	负责搜集、保存系统中所有的音频采集设备信息
CDSDevice	2-3	保存一个设备的属性（包括名字、类型等）
CDeviceObserver	2-3-1	负责监测视频采集设备的热插拔
CMsgReceiver	2-6	自定义消息的接收者的基类

（续表）

类名	所在目录	功能描述
CMsgStation	2-6	自定义消息的发送者的基类
CObjectList	2-6	一个列表类（从CPtrArray类派生，列表元素为指针）
CCrossbar	2-6	负责视频采集设备输入端子的相关操作
CDXGraph	2-1	封装了Filter Graph的控制功能
CDXFilter	2-1	Filter的包装类，并作为具体Filter包装类的父类
CVideoWindow	2-1	视频显示窗口（从CWnd类派生）
UDsUtils	2-1	工具类，完成了DirectShow相关的一系列相对独立的功能（如枚举系统设备、创建设备Filter、显示Filter的属性页等）
UFilterUtils	2-1	工具类，完成了Filter相关的一系列相对独立的功能（如得到某个Filter上的输入Pin、输出Pin、未连接的Pin、输入输出Pin的个数等）
CAudioCaptureFilter	2-2	音频采集设备的包装类（负责设备的创建、参数设置）
CVideoCaptureFilter	2-2	视频采集设备的包装类
CDVCaptureFilter	2-2	数码摄像机的包装类
CAudioRendererFilter	2-2	Audio Renderer Filter的包装类（用于声音播放）
CAviMuxFilter	2-2	AVI Mux Filter的包装类（用于音视频合成）
CFileWriterFilter	2-2	File Writer Filter的包装类（用于写文件）
CImageGrabberFilter	2-2	HQ Image Grabber Filter的包装类（用于即时抓图）
COverlayMixerFilter	2-2	Overlay Mixer Filter的包装类（用于视频显示）
CSmartTeeFilter	2-2	Smart Tee Filter的包装类（用于分路）
CVideoRendererFilter	2-2	Video Renderer Filter的包装类（用于视频显示，并且控制传统的Video Renderer、VMR-7和VMR-9使用的优先顺序）

上述这些类中，需要重点理解的有CDXGraph、CDXFilter、CVideoWindow、UDsUtils、UFilterUtils等。它们都是可复用的，在开发其他DirectShow应用程序时可以直接使用。另外，这里还要提醒一下的是：在MFC程序中，当视频窗口嵌入到主窗口的一个子窗口中时，如果处理不当，经常会碰到视频窗口的刷新问题（视频窗口出现黑屏）。解决的方法其实也很简单：改变视频窗口的风格为WS_CLIPCHILDREN，并且在主界面类的WM_ERASEBKGND消息的响应函数中做如下一些处理：

```
//
// CVideoWindow.cpp
//
void CVideoWindow::PreSubclassWindow(void)
{
    // 修改视频窗口风格为WS_CLIPCHILDREN
    ModifyStyle(0, WS_CLIPCHILDREN);
    CWnd::PreSubclassWindow();
}
```

```
// AVCapDlg.h : header file
//
#include " CVideoWindow.h"
class CAVCapDlg : public CDialog
{
public:
// Dialog Data
    CVideoWindow    mVideoWindow;
protected:
    //{{AFX_MSG(CAVCapDlg)
    afx_msg BOOL OnEraseBkgnd(CDC* pDC);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

// AVCapDlg.cpp : implementation file
//
void CAVCapDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CAVCapDlg)
    DDX_Control(pDX, IDC_VIDEO_WINDOW, mVideoWindow);
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAVCapDlg, CDialog)
    //{{AFX_MSG_MAP(CAVCapDlg)
    ON_WM_ERASEBKGND()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

// WM_ERASEBKGND消息的响应函数
BOOL CAVCapDlg::OnEraseBkgnd(CDC* pDC)
{
    CRect rc;
    mVideoWindow.GetWindowRect(&rc);
    ScreenToClient(&rc);
    // 抠除视频窗口区域
    pDC->ExcludeClipRect(&rc);
    return CDialog::OnEraseBkgnd(pDC);
}
```

AVCap程序使用到的一些重要的类的继承结构如图2.40所示。

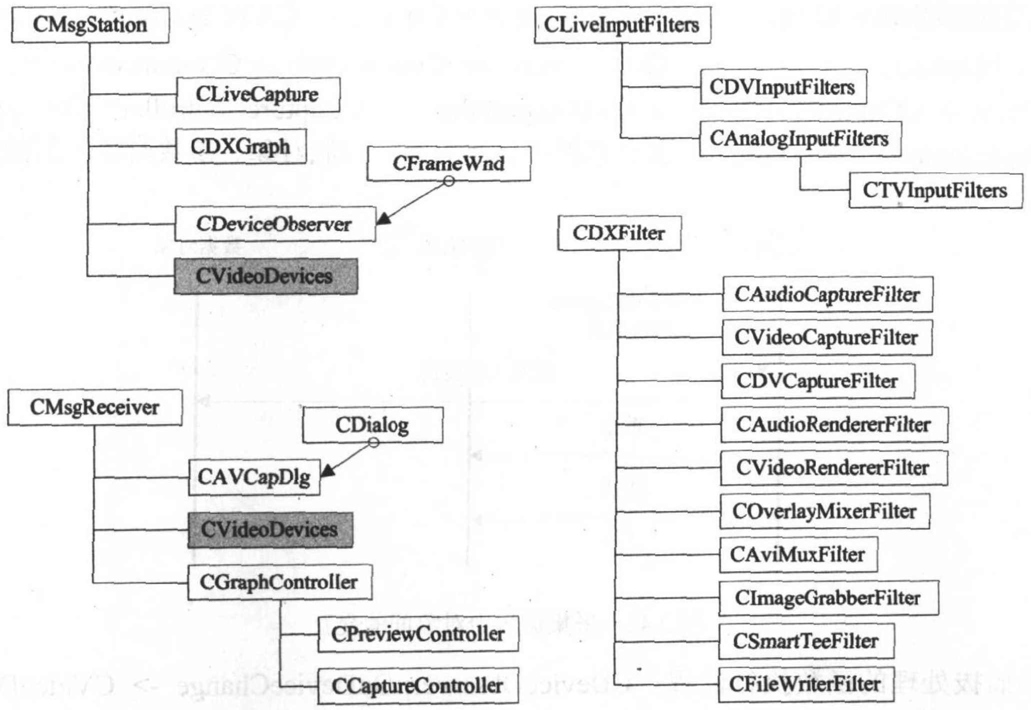


图 2.40 AVCap 程序的类继承结构

2.4.3 流程概要

预览的函数执行流程：CAVCapDlg::OnInitDialog -> CAVCapDlg::CreateController -> CgraphController::Activate -> CGraphController::CreateGraph -> CGraphController::Create InputFilters -> CPreviewController::CreateOutputFilters -> CPreviewController:: ConnectFilters -> CGraphController::PlayGraph。这个流程中界面对象、控制对象、参数对象之间的交互如图2.41所示。

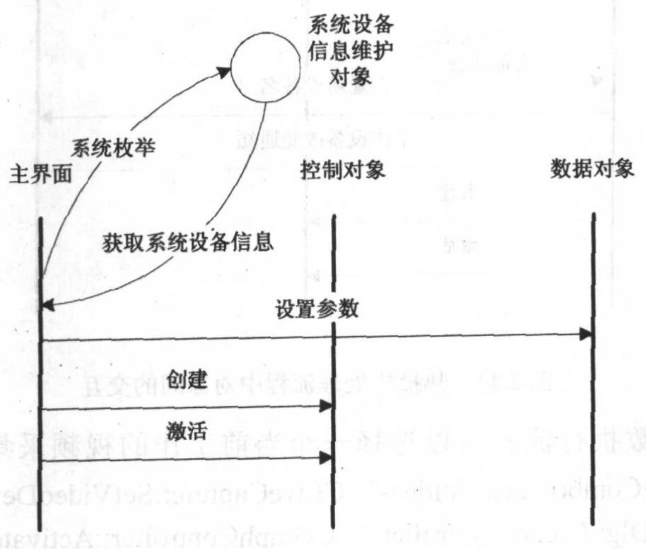


图 2.41 预览流程中对象间的交互

采集的函数执行流程：CAVCapDlg::OnButtonCapture -> CAVCapDlg::CreateController -> CGraphController::Activate -> CGraphController::CreateGraph -> CGraphController:: Create InputFilters -> CCaptureController::CreateOutputFilters -> CCaptureController:: ConnectFilters -> CGraphController::PlayGraph。这个流程中界面对象、控制对象、参数对象之间的交互如图2.42所示。

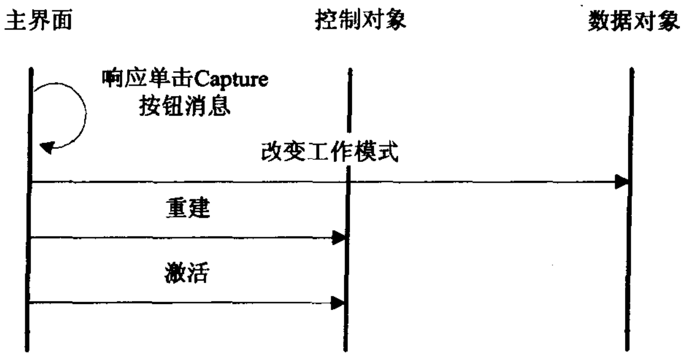


图 2.42　采集流程中对象间的交互

热插拔处理的函数执行流程：CDeviceObserver::OnDeviceChange -> CVideoDevices:: ReceiveMessage -> CAVCapDlg::ReceiveMessage -> CAVCapDlg::FillVideoDevices -> Clive Capture::SetVideoDevice -> CAVCapDlg::Respond -> CAVCapDlg::CreateController -> CgraphController::Activate。这个流程中界面对象、控制对象、参数对象之间的交互如图2.43所示。

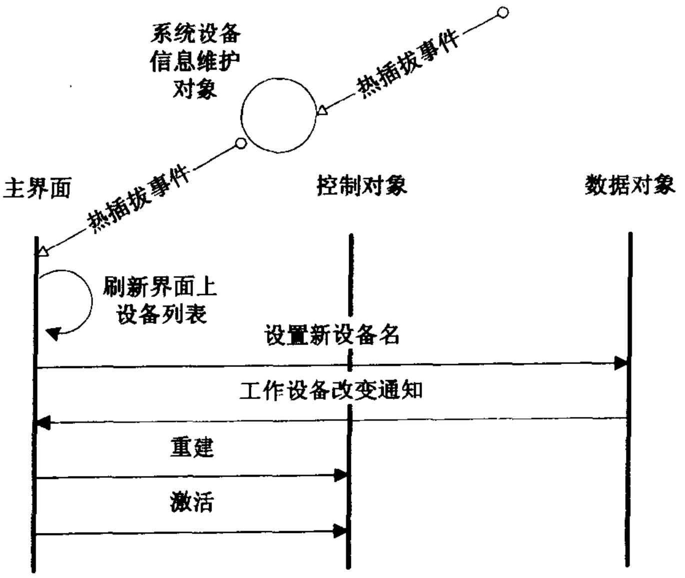


图 2.43　热插拔处理流程中对象间的交互

参数设置的函数执行流程（以选择一个当前工作的视频采集设备为例）：CAV CapDlg::OnSelchangeComboDeviceVideo-> CLiveCapture::SetVideoDevice -> CAVCapDlg:: Respond -> CAVCapDlg::CreateController -> CGraphController::Activate。这个流程中界面对象、控制对象、参数对象之间的交互如图2.44所示。

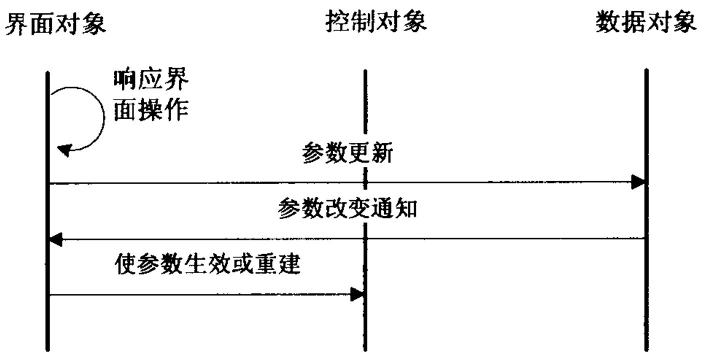


图 2.44 参数设置流程中对象间的交互

提示： 本书配套光盘的SourceCodes\Chapter02\AVCap目录下提供了AVCap程序的完整实现。打开AVCap.dsw文件可以同时浏览AVCap和FilterImageGrabber两个项目。

网络应用

3.1 网络编程基础

多媒体技术与网络技术的结合，使得网络生活变得多姿多彩。姑且认为DirectShow是单机的多媒体技术，一旦融合了网络技术，DirectShow更显现了它强大的生命力。本章将着重介绍DirectShow技术在网络方面的应用。

网络编程，当然要用到Windows Socket（套接字）技术。Socket相关的操作由一系列API函数来完成，比如socket、bind、listen、connect、accept、send、sendto、recv、recvfrom等。调用这些API函数有一定的先后次序，有些函数的参数还比较复杂，对于开发者来说不是很好用。于是，微软的MFC提供了两个类：CAsyncSocket和CSocket，极大地方便了Socket功能的使用。这两个类的继承关系如图3.1所示。

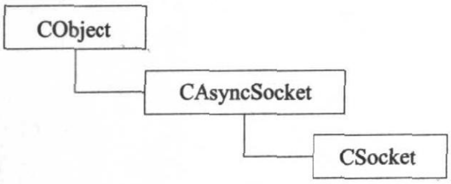


图 3.1　MFC Socket 类的继承关系

CAsyncSocket类在较低层次上封装了Windows Socket API，并且通过内建一个（隐藏的）窗口，实现了适合Windows应用的异步机制（Windows Socket API默认情况下工作在阻塞模式，不方便直接在消息驱动的Windows程序上使用）。CSocket类从CAsyncSocket类派生，进一步简化了Socket功能的应用。不过很遗憾，正因为这两个类都内建了一个窗口，它们并不是线程安全的（thread-safe）；如果要在多线程环境下应用Socket功能，建议自行封装

Socket API函数。

使用Socket传输数据主要有两种方式：TCP传输和UDP传输。（OSI参考模型将网络通信分成7个层次，从下往上依次为物理层、数据链路层、网络层、传输层、会话层、表示层、应用层；TCP和UDP均是传输层的协议。）下面，分别介绍这两种数据传输方式。

提示：本章在介绍网络通信双方的时候，会使用两组关键词：服务器-客户机和本地端-远程端。其中，服务器-客户机是根据角色来界定的；而本地端-远程端是一个相对概念，依据不同的参照物，可以分别表示不同的角色。比如以服务器为参照物，可以称服务器为本地端，称客户机为远程端；而如果以客户机为参照物，可以称客户机为本地端，称服务器为远程端。

3.1.1 TCP传输

TCP（Transmission Control Protocol，传输控制协议）是一种面向连接的网络传输协议。TCP协议的特点是支持多数据流操作，提供流控和错误控制，甚至能完成对乱序到达报文的重新排序等。因此，TCP提供了可靠的应用数据传输服务。

通信双方使用TCP传输的一般过程如图3.2所示。

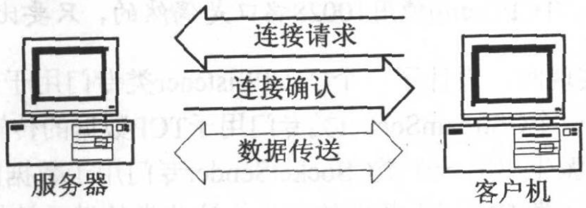


图3.2 TCP通信的一般过程

本节将要实现一个TCP传输的演示程序TCPDemo，它包括服务器和客户机两个部分。它们的程序界面如图3.3所示。

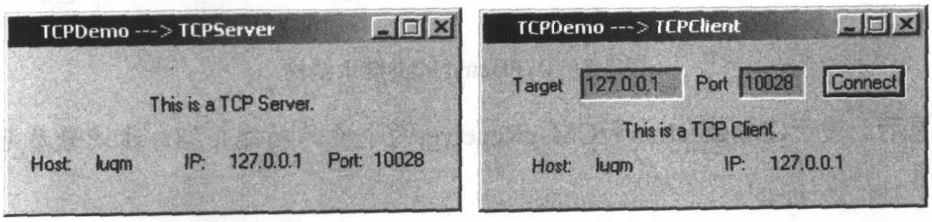


图3.3 TCP传输演示程序界面

TCPDemo的演示过程如下：

（1）将服务器和客户机两部分程序都运行起来（此时服务器已经启动了侦听客户机连接请求的子线程，侦听端口号为10028）。

（2）在客户机程序界面上输入服务器的IP地址（如果服务器和客户机运行在同一台机器上，IP地址可以指定为127.0.0.1）、侦听端口号（因为服务器在10028端口上侦听，这里也应该指定为10028）。

（3）单击客户机程序界面上的Connect按钮，向服务器发送Socket连接请求。

（4）服务器侦听到有客户机的连接请求后便接受它（于是在两个程序之间就建立了一条可靠的Socket连接），然后服务器会向客户机发送两次字符串数据。

（5）客户机接收到数据后，弹出两次如图3.4所示的消息框。

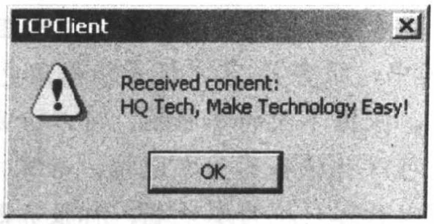


图 3.4　TCP 传输客户机接收到数据后显示的消息框

提示：TCPDemo为什么使用10028作为TCP通信的端口号？因为TCP数据包的TCP头结构中，使用了16位的域来表示一个端口号，因此，有65536个可能的端口号。0~1023是周知口（众所周知的端口，如80是超文本传输协议http的端口，25是简单邮件传输协议smtp的端口，20和21是文件传输协议ftp的端口等），比1023大的端口号通常被称为高端口号。应用程序一般使用高端口号提供自己的通信服务。TCPDemo使用10028端口是偶然的，只要比1023大就可以了。

TCPDemo在具体实现时，设计了一个CTCPListener类专门用于服务器对特定TCP端口的侦听。另外，设计了一个CStreamSocket类专门用于TCP数据的传输。CStreamSocket作为基类，服务器程序从它派生出另一个类CSocketSender专门用于数据的发送，客户机程序从它派生出CSocketReceiver类专门用于数据的接收。这些类的继承结构如图3.5所示。

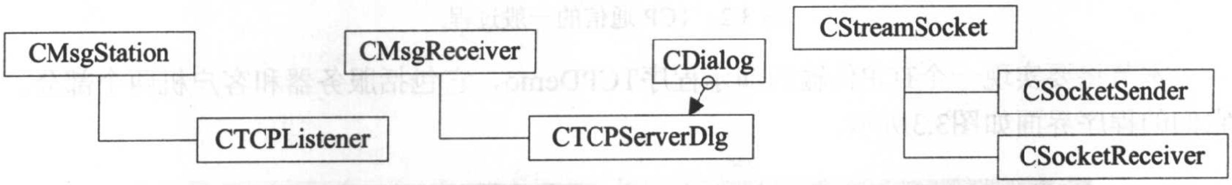


图 3.5　TCPDemo 的类继承结构

提示：关于CMsgStation和CMsgReceiver两个类的功能介绍，请读者参见2.4.1节。

```
//
// CTCPListener.h
//

#ifndef __H_CTCPListener__
#define __H_CTCPListener__

#include "CMsgStation.h"

class CTCPListener : public CMsgStation
{
```

```
protected:
    SOCKET      mListener;      // 用于侦听的Socket
    SOCKET      mAccepted;      // 用于与远程端建立连接的Socket
    WORD        mListenPort;    // 侦听端口号
    BOOL        mIsListening;   // 是否正在侦听的标记
    HANDLE      mLsnThread;     // 侦听线程
public:
    CTCPListener();
    virtual ~CTCPListener();
public:
    // 设置/得到侦听的端口号
    void SetListenPort(WORD inPort);
    WORD GetListenPort(void);
    // 创建/销毁用于侦听的Socket
    BOOL Create(void);
    void DeleteListener(void);
    // 销毁服务器与客户机建立连接的Socket
    void DeleteAccepted(void);
    // 启动/停止侦听线程
    BOOL StartListening(void);
    void StopListening(void);
    // 得到服务器与客户机建立连接的Socket（用于数据传输）
    SOCKET GetAccepted(void);
private:
    BOOL Accept(void); // 接受远程端的连接请求
    static DWORD WINAPI ListeningThrd(void *pParam);  // 侦听线程执行体
};

#endif // __H_CTCPListener__

//
// CTCPListener.cpp
//
#include "stdafx.h"
#include "CTCPListener.h"
#include "Netdefs.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
//////////////////////////////////////////////////////////////////////////
CTCPListener::CTCPListener()
{
    // 参数初始化
    mListener = INVALID_SOCKET;
    mAccepted = INVALID_SOCKET;
    // 默认在10028端口上侦听
    mListenPort     = 10028;
    mLsnThread      = NULL;
    mIsListening    = FALSE;
```

```
}
CTCPListener::~CTCPListener()
{
    // 销毁Socket
    DeleteAccepted();
    DeleteListener();
    // 停止侦听线程
    StopListening();
}
// 设置侦听端口号
void CTCPListener::SetListenPort(WORD inPort)
{
    mListenPort = inPort;
}
// 得到侦听端口号
WORD CTCPListener::GetListenPort(void)
{
    return mListenPort;
}
// 创建用于侦听的Socket
BOOL CTCPListener::Create(void)
{
    DeleteListener();  // 销毁侦听Socket

    int  val  = 0;
    BOOL pass = FALSE;
    // 创建一个TCP传输的Socket
    mListener = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (mListener != INVALID_SOCKET)
    {
        // 在Socket上进行参数设置
        BOOL  sopt = TRUE;
        setsockopt(mListener, IPPROTO_TCP, TCP_NODELAY,
            (char *)&sopt, sizeof(BOOL));
        // 在销毁Socket时不必等待未发送完的数据完全发送出去
        setsockopt(mListener, SOL_SOCKET, SO_DONTLINGER,
            (char *)&sopt, sizeof(BOOL));
        // 绑定Socket到指定的侦听端口
        SOCKADDR_IN  addr;
        memset(&addr, 0, sizeof(SOCKADDR_IN));
        addr.sin_family = AF_INET;
        addr.sin_addr.s_addr  = htonl(INADDR_ANY);
        addr.sin_port    = htons(mListenPort);
        val = bind(mListener, (struct sockaddr*) &addr, sizeof(addr));
        pass = (val != SOCKET_ERROR);
    }
    if (pass)
    {
        // 将Socket置于侦听状态
        val = listen(mListener, SOMAXCONN);
        pass = (val != SOCKET_ERROR);
    }
```

```
    if (!pass)
    {
        DeleteListener();
    }
    return pass;
}
// 销毁用于侦听的Socket
void CTCPListener::DeleteListener(void)
{
    if (mListener != INVALID_SOCKET)
    {
        closesocket(mListener);
        mListener = INVALID_SOCKET;
    }
}
// 销毁服务器与客户机建立连接的Socket
void CTCPListener::DeleteAccepted(void)
{
    if (mAccepted != INVALID_SOCKET)
    {
        closesocket(mAccepted);
        mAccepted = INVALID_SOCKET;
    }
}
// 启动侦听线程（因为用于接受连接请求的accept函数调用时会阻塞）
BOOL CTCPListener::StartListening(void)
{
    // 如果侦听Socket没有创建，则创建它
    if (mListener == INVALID_SOCKET)
    {
        Create();
    }
    if (mListener != INVALID_SOCKET)
    {
        if (mIsListening)
        {
            return TRUE;
        }
        // 启动侦听线程
        DWORD threadID = 0;
        mLsnThread = CreateThread(NULL, 0, ListeningThrd,
            this, 0, &threadID);
        return (mLsnThread != NULL);
    }
    return FALSE;
}
// 停止侦听线程
void CTCPListener::StopListening(void)
{
    if (mListener != INVALID_SOCKET && mIsListening)
    {
        // 销毁侦听Socket，于是accept将脱离阻塞状态
```

```
        DeleteListener();
        // 等待侦听线程完全退出
        if (mLsnThread != NULL)
        {
            WaitForSingleObject(mLsnThread, INFINITE);
            mLsnThread = NULL;
        }
    }
}
// 接受远程端的连接请求（创建一个新的Socket用于与远程端建立一条连接）
BOOL CTCPListener::Accept(void)
{
    if (mListener != INVALID_SOCKET)
    {
        SOCKADDR_IN  saddr;
        int len = sizeof(SOCKADDR_IN);
        // 侦听远程端的连接请求（如果没有连接请求，这个函数将阻塞）
        SOCKET accepted = accept(mListener, (SOCKADDR *)&saddr, &len);
        if (accepted == INVALID_SOCKET)
        {
            return FALSE;
        }
        // 注意：目前仅支持建立一条Socket连接
        // 在建立新的连接之前将以前的连接断开
        DeleteAccepted();
        // 保存与远程端建立连接的Socket
        mAccepted = accepted;
        // 在Socket上设置一些参数
        BOOL sopt = TRUE;
        setsockopt(mAccepted, IPPROTO_TCP, TCP_NODELAY,
            (char *)&sopt, sizeof(BOOL));
        setsockopt(mAccepted, SOL_SOCKET, SO_DONTLINGER,
            (char *)&sopt, sizeof(BOOL));
        return TRUE;
    }
    return FALSE;
}
// 当与远程端连接的Socket取出之后，保存该Socket的变量置为无效
// 取出的Socket由取出者负责销毁
SOCKET CTCPListener::GetAccepted(void)
{
    SOCKET ret = mAccepted;
    mAccepted  = INVALID_SOCKET;
    return ret;
}
// 侦听线程的函数执行体
DWORD WINAPI CTCPListener::ListeningThrd(void *pParam)
{
    ASSERT(pParam);
    // 获得侦听对象指针
    CTCPListener * pListen = (CTCPListener *) pParam;
    pListen->mIsListening  = TRUE;
```

```
    while (pListen->mIsListening)
    {
        // 开始侦听（如果没有远程端发送连接请求，这个函数将阻塞）
        if (!pListen->Accept())
        {
            pListen->mIsListening = FALSE;
            break;
        }
        else
        {
            // const long cNewSocketAccepted  = 6688;
            // 发送给上层观察者一个自定义消息cNewSocketAccepted，表示一条Socket
            // 连接已经建立（可以用它进行数据传输了！）
            pListen->Broadcast(cNewSocketAccepted);
        }
    }

    return 1;
}

//
// CStreamSocket.h
//

#ifndef __H_CStreamSocket__
#define __H_CStreamSocket__

class CStreamSocket
{
protected:
    SOCKET      mSocket;        // 用于数据发送或接收的Socket
    BOOL        mIsConnected;   // Socket是否已经建立了连接的标记
    BOOL        mIsReceiving;   // 使用独立的线程进行数据接收
    HANDLE      mRcvThread;
    BOOL        mIsSending;     // 使用独立的线程进行数据发送
    HANDLE      mSndThread;
public:
    CStreamSocket();
    virtual ~CStreamSocket();
public:
    BOOL Attach(SOCKET inSocket);   // 关联一个Socket
    void Detach(void);              // 销毁Socket
    // 向指定IP地址、端口号的机器发送连接请求
    BOOL ConnectTo(const char * inTarget, WORD inPort);
    BOOL IsConnected(void) { return mIsConnected; };
    // 用于数据接收的控制函数
    BOOL StartReceiving(void);
    void StopReceiving(void);
    BOOL IsReceiving(void) { return mIsReceiving; };
    // 用于数据发送的控制函数
    BOOL StartSending(void);
    void StopSending(void);
```

```
    BOOL IsSending(void) { return mIsSending; };
protected:
    static DWORD WINAPI ReceivingThrd(void * pParam);  // 接收线程执行体
    static DWORD WINAPI SendingThrd(void * pParam);    // 发送线程执行体
    // 接收/发送数据循环过程（虚函数，供子类定制）
    virtual void ReceivingLoop(void);
    virtual void SendingLoop(void);
};

#endif // __H_CStreamSocket__

//
// CStreamSocket.cpp
//

#include "stdafx.h"
#include "CStreamSocket.h"
#include "UNetwork.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
//////////////////////////////////////////////////////////////////////////
CStreamSocket::CStreamSocket()
{
    // 参数初始化
    mSocket         = INVALID_SOCKET;
    mIsConnected    = FALSE;
    mIsReceiving    = FALSE;
    mIsSending      = FALSE;
    mRcvThread      = NULL;
    mSndThread      = NULL;
}
// 销毁Socket，停止发送/接收线程
CStreamSocket::~CStreamSocket()
{
    Detach();
    StopSending();
    StopReceiving();
}
// 关联一个Socket到本包装对象
BOOL CStreamSocket::Attach(SOCKET inSocket)
{
    // 如果已经包装了一个Socket，则返回一个错误值
    if (mSocket != INVALID_SOCKET)
    {
        return FALSE;
    }
    // 保存Socket句柄
    mSocket     = inSocket;
```

```
    mIsConnected = TRUE;
    return TRUE;
}
// 销毁Socket
void CStreamSocket::Detach(void)
{
    if (mSocket != INVALID_SOCKET)
    {
        closesocket(mSocket);
        mSocket = INVALID_SOCKET;
        mIsConnected = FALSE;
    }
}
// 向指定IP地址、端口号的机器发送连接请求
BOOL CStreamSocket::ConnectTo(const char * inTarget, WORD inPort)
{
    if (mIsConnected)
    {
        return TRUE;
    }
    // 首先创建一个TCP传输的Socket
    mSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (mSocket != INVALID_SOCKET)
    {
        // 在成功创建的Socket上调整参数
        BOOL sopt = TRUE;
        setsockopt(mSocket, IPPROTO_TCP, TCP_NODELAY,
            (char *)&sopt, sizeof(BOOL));
        setsockopt(mSocket, SOL_SOCKET, SO_DONTLINGER,
            (char *)&sopt, sizeof(BOOL));
        // 向服务器发送连接请求
        SOCKADDR_IN  saddr;
        memset(&saddr, 0, sizeof(SOCKADDR_IN));
        saddr.sin_addr.S_un.S_addr  = inet_addr(inTarget);
        saddr.sin_family            = AF_INET;
        saddr.sin_port              = htons((WORD)inPort);
        if (connect(mSocket, (SOCKADDR *)&saddr, sizeof(SOCKADDR_IN)) != 0)
        {
            // 跟踪Socket错误
#ifdef _DEBUG
            UNetwork::DumpSocketError();
#endif
            // 如果连接失败，则销毁刚才创建的Socket
            Detach();
            return FALSE;
        }
        mIsConnected = TRUE;
        return TRUE;
    }
    return FALSE;
}
// 启动数据接收线程（因为Socket数据接收函数调用时会阻塞）
```

```
BOOL CStreamSocket::StartReceiving(void)
{
    if (mSocket != INVALID_SOCKET)
    {
        if (mIsReceiving)
        {
            return TRUE;
        }
        DWORD threadID = 0;
        mRcvThread = CreateThread(NULL, 0, ReceivingThrd,
            this, 0, &threadID);
        return (mRcvThread != NULL);
    }
    return FALSE;
}
// 停止数据接收线程
void CStreamSocket::StopReceiving(void)
{
    if (mIsReceiving)
    {
        // 销毁Socket，使接收函数失败或脱离阻塞
        Detach();
        // 等待数据接收线程的完全退出
        if (mRcvThread != NULL)
        {
            WaitForSingleObject(mRcvThread, INFINITE);
            mRcvThread = NULL;
        }
    }
}
// 启动数据发送线程（以提高数据发送的效率）
BOOL CStreamSocket::StartSending(void)
{
    if (mSocket != INVALID_SOCKET)
    {
        if (mIsSending)
        {
            return TRUE;
        }
        DWORD threadID = 0;
        mSndThread = CreateThread(NULL, 0, SendingThrd,
            this, 0, &threadID);
        return (mSndThread != NULL);
    }
    return FALSE;
}
// 停止数据发送线程
void CStreamSocket::StopSending(void)
{
    if (mIsSending)
    {
        // 销毁Socket，使发送函数失败或脱离阻塞
```

```
        Detach();
        if (mSndThread != NULL)
        {
            // 等待数据发送线程的完全退出
            WaitForSingleObject(mSndThread, INFINITE);
            mSndThread = NULL;
        }
    }
}
// 数据接收线程的函数执行体
DWORD WINAPI CStreamSocket::ReceivingThrd(void * pParam)
{
    CStreamSocket * pSock = (CStreamSocket *) pParam;
    if (pSock)
    {
        pSock->mIsReceiving = TRUE;
        // 执行接收循环
        pSock->ReceivingLoop();
        return 1;
    }
    return 0;
}
// 数据发送线程的函数执行体
DWORD WINAPI CStreamSocket::SendingThrd(void * pParam)
{
    CStreamSocket * pSock = (CStreamSocket *) pParam;
    if (pSock)
    {
        pSock->mIsSending = TRUE;
        // 执行发送循环
        pSock->SendingLoop();
        return 1;
    }
    return 0;
}
// 虚函数，供子类定制实际的数据接收（循环）过程
void CStreamSocket::ReceivingLoop(void)
{
}
// 虚函数，供子类定制实际的数据发送（循环）过程
void CStreamSocket::SendingLoop(void)
{
}

//
// CSocketSender.h
//

#ifndef __H_CSocketSender__
#define __H_CSocketSender__

#include "CStreamSocket.h"
```

```
class CSocketSender : public CStreamSocket
{
public:
    CSocketSender();
    virtual ~CSocketSender();
protected:
    virtual void SendingLoop(void);  // 定制数据发送过程
};

#endif // __H_CSocketSender__

//
// CSocketSender.cpp
//
// 服务器程序定制的数据发送过程
void CSocketSender::SendingLoop(void)
{
    char  buf[1024];  // 发送数据使用的缓存
    int   bytes = 0;
    // 定义一个字符串作为发送的数据内容
    char  str[] = "HQ Tech, Make Technology Easy!";
    // 发送数据的总长度 = 字符串长度 + 头信息长度
    int   len   = strlen(str) + sizeof(Net_Header);
    // 在数据内容之前加上一个自定义头信息（用以说明数据内容的长度）
    Net_Header * pHeader = (Net_Header *) buf;
    pHeader->pack_size   = strlen(str);
    pHeader->my_hton(); // 字节顺序转换
    // 将欲发送的数据内容和头信息整合
    strcpy(buf+sizeof(Net_Header), str);
    // 作为演示，将上述定义的字符串数据发送两次
    int   counter = 2;
    while (mIsSending)
    {
        // 使用Socket进行一次数据发送
        bytes = send(mSocket, buf, len, 0);
        if (bytes == SOCKET_ERROR)
        {
            Detach();
            mIsSending = FALSE;
            break;
        }
        // 当完成两次发送后断开Socket连接，结束发送线程
        if (--counter == 0)
        {
            Detach();
            mIsSending = FALSE;
            break;
        }
    }
}
//
```

```
// CSocketReceiver.h
//

#ifndef __H_CSocketReceiver__
#define __H_CSocketReceiver__

#include "CStreamSocket.h"

class CSocketReceiver : public CStreamSocket
{
public:
    CSocketReceiver();
    virtual ~CSocketReceiver();
protected:
    virtual void ReceivingLoop(void); // 定制数据接收过程
};

#endif // __H_CSocketReceiver__

//
// CSocketReceiver.cpp
//
// 客户机程序定制的数据接收过程
void CSocketReceiver::ReceivingLoop(void)
{
    // 接收数据使用的缓存
    char  buf[1024];
    int   bytes = 0;
    Net_Header * pHeader = (Net_Header *) buf;

    while (mIsReceiving)
    {
        // 首先接收一个头信息（头信息内包含了随后的有效数据长度）
        bytes = recv(mSocket, buf, sizeof(Net_Header), 0);
        if (bytes == SOCKET_ERROR || bytes == 0)
        {
            Detach();
            mIsReceiving = FALSE;
            break;
        }

        pHeader->my_ntoh();  // 字节顺序转换
        // 继续读取后续的有效数据（即一个字符串内容）
        bytes = recv(mSocket, buf, pHeader->pack_size, 0);
        if (bytes == SOCKET_ERROR || bytes == 0)
        {
            Detach();
            mIsReceiving = FALSE;
            break;
        }

        buf[bytes]  = '\0';
```

```
        // 弹出一个消息框显示接收到的字符串内容
        CString msg = "Received content:\n";
        AfxMessageBox(msg + buf);
    }
}
```

那么，TCPDemo是怎么使用CTCPListener、CStreamSocket、CSocketSender、Csocket-Receiver这几个类的呢？先来看服务器程序TCPServer。这是一个基于对话框的MFC程序。它在对话框类CTCPServerDlg中定义了两个成员：一个是CTCPListener类的实例，一个是CSocketSender类的实例。前者用于侦听客户机的连接请求，后者负责实际的Socket数据发送。然后，在主对话框的初始化函数中创建用于侦听的Socket，并启动侦听线程。当有客户机发出连接请求，并且服务器成功接受后，就启动数据发送线程真正开始数据的发送。

```
// TCPServerDlg.h : header file
//
class CTCPServerDlg : public CDialog, public CMsgReceiver
{
public:
    CTCPServerDlg(CWnd* pParent = NULL);
protected:
    HICON m_hIcon;

    CTCPListener    mListener;   // 用于侦听客户机的连接请求
    CSocketSender   mNetSender;  // 用于数据发送
    // 自定义消息的处理函数
    virtual bool ReceiveMessage(MessageT inMessage, void * ioParam,
        void * ioParam2);
    // 其他成员定义（省略）
    // ……
};

// TCPServerDlg.cpp : implementation file
//
CTCPServerDlg::CTCPServerDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CTCPServerDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CTCPServerDlg)
    mHostPort = 10028;
    //}}AFX_DATA_INIT
    // Note that LoadIcon does not require a subsequent DestroyIcon in Win32
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}
// 主对话框的初始化函数
BOOL CTCPServerDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // Set the icon for this dialog.  The framework does this automatically
    //  when the application's main window is not a dialog
    SetIcon(m_hIcon, TRUE);         // Set big icon
    SetIcon(m_hIcon, FALSE);        // Set small icon
```

```
    // 获取本地机器的IP地址、机器名，并在界面上显示
    char hostName[100];
    char hostIP[50];
    if (UNetwork::GetHostInfo(hostIP, hostName))
    {
        mEditHostName.SetWindowText(hostName);
        mEditHostIP.SetWindowText(hostIP);
    }
    // 主界面对象是mListener对象的观察者（因为它想获得Socket连接建立的通知）
    mListener.AddMsgReceiver(this);
    // 设置侦听端口号
    mListener.SetListenPort(mHostPort);
    // 创建侦听Socket，成功后启动一个侦听线程
    if (mListener.Create())
    {
        mListener.StartListening();
    }

    return TRUE;  // return TRUE  unless you set the focus to a control
}
// 当接收到Socket连接已经建立的通知后，启动数据发送线程向客户机发送数据
bool CTCPServerDlg::ReceiveMessage(MessageT inMessage, void * ioParam, void
    * ioParam2)
{
    if (inMessage == cNewSocketAccepted)
    {
        // 获取建立连接的Socket
        mNetSender.Attach(mListener.GetAccepted());
        // 启动数据发送线程
        mNetSender.StartSending();
        return true;
    }

    return CMsgReceiver::ReceiveMessage(inMessage, ioParam, ioParam2);
}
```

提示：使用MFC开发Socket程序时，一般要包含afxsock.h头文件（可以加在stdafx.h文件中）。程序运行之前，还要调用AfxSocketInit函数进行Socket函数库的初始化，实现如下：

```
//
// TCPServer.cpp
//
BOOL CTCPServerApp::InitInstance()
{
    // --- Socket函数库的初始化 ---
    // AfxSocketInit内部调用WSAStartup函数，
    // 并且能够保证在程序退出之前自动调用WSACleanup函数！
    if (!AfxSocketInit())
    {
        AfxMessageBox("Socket initializing failded!");
```

```
            return FALSE;
        }

        // 创建主对话框
        CTCPServerDlg dlg;
        m_pMainWnd = &dlg;
        int nResponse = dlg.DoModal();
        if (nResponse == IDOK)
        {
        }
        else if (nResponse == IDCANCEL)
        {
        }
        return FALSE;
    }
```

再来看客户机程序TCPClient的实现。这也是一个基于对话框的MFC程序。它在对话框类CTCPClientDlg中定义了一个是CSocketReceiver类的实例，专门用于向服务器发出连接请求，以及接收服务器发送过来的数据。

```
// TCPClientDlg.h : header file
//
class CTCPClientDlg : public CDialog
{
public:
    CTCPClientDlg(CWnd* pParent = NULL);
protected:
    HICON m_hIcon;
    CSocketReceiver    mNetReceiver;  // 用于数据接收
    // 其他成员定义（省略）
    // ……
};

// TCPClientDlg.cpp : implementation file
//
CTCPClientDlg::CTCPClientDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CTCPClientDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CTCPClientDlg)
    mTargetIP   = _T("127.0.0.1");
    mTargetPort = 10028;
    //}}AFX_DATA_INIT
    // Note that LoadIcon does not require a subsequent DestroyIcon in Win32
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}
// 主对话框的初始化函数
BOOL CTCPClientDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    // Set the icon for this dialog.  The framework does this automatically
    //  when the application's main window is not a dialog
    SetIcon(m_hIcon, TRUE);         // Set big icon
```

```
    SetIcon(m_hIcon, FALSE);        // Set small icon

    // 获取本地机器的IP地址、机器名，并在界面上显示
    char hostName[100];
    char hostIP[50];
    if (UNetwork::GetHostInfo(hostIP, hostName))
    {
        mEditHostName.SetWindowText(hostName);
        mEditHostIP.SetWindowText(hostIP);
    }

    return TRUE;  // return TRUE  unless you set the focus to a control
}
// 界面上的Connect按钮的响应函数
void CTCPClientDlg::OnButtonConnect()
{
    // 从界面上获取最新的数据
    UpdateData(TRUE);
    // 向指定IP地址、端口号的服务器发出连接请求
    if (mNetReceiver.ConnectTo(mTargetIP, mTargetPort))
    {
        // 连接成功后启动一个子线程用于数据接收
        mNetReceiver.StartReceiving();
    }
    else
    {
        // 弹出连接失败的消息框
        CString msg;
        msg.Format("Connecting to %s:%d failed!", mTargetIP, mTargetPort);
        AfxMessageBox(msg);
    }
}
```

TCPServer和TCPClient两个程序的整个交互过程如图3.6所示。

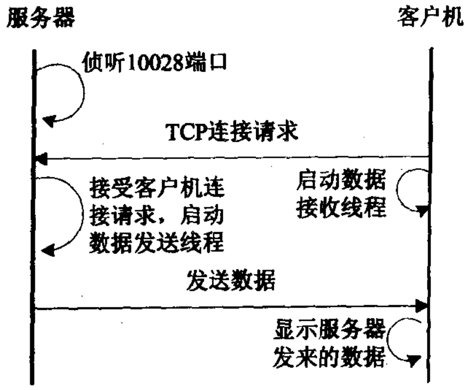


图 3.6　TCPServer 和 TCPClient 的交互过程

值得注意的是，TCP传输的是一种字节流数据。在应用程序中，有时需要对这些数据进行一些控制，或者获得一些说明。于是，TCPServer和TCPClient两个程序在进行TCP通信

的时候，定义了一个简单的应用协议，即TCPServer发出的数据总是使用一个“负载头+负载数据”的结构。其中，负载数据是真正需要传输的数据内容，而负载头是对负载数据的一个说明，指示了负载数据的实际长度。这个负载头的定义如下：

```
//
// Netdefs.h
//
struct Net_Header
{
    unsigned long pack_size;  // 使用一个无符号整型变量，说明负载数据的长度

    // 将本结构的变量值从主机字节顺序转换成网络字节顺序
    void my_hton(void)
    {
        pack_size   = htonl(pack_size);
    };
    // 将本结构的变量值从网络字节顺序转换成主机字节顺序
    void my_ntoh(void)
    {
        pack_size   = ntohl(pack_size);
    };
};
```

小知识：字节顺序

很少有人关心字节顺序（Byte Ordering），因为它真的很少用到。何为字节顺序呢？让我们先来看一个例子，假设现在有一个WORD类型的变量，它的值为0x7788，那么它在内存中是怎么存放的呢？

事实上，对于不同的CPU、不同的操作系统，图3.7中的两种字节顺序都是可能的。如果像图3.7的左图那样：高字节在前，低字节在后，则这种字节顺序称为big-endian；如果像图3.7的右图那样：低字节在前，高字节在后，则这种字节顺序称为little-endian。常见的CPU、操作系统上使用的字节顺序如表3.1所示。

内存中：	低地址	高地址	低地址	高地址
	0x77	0x88	0x88	0x77
	big-endian		little-endian	

图 3.7　两种字节顺序

表3.1　常见的CPU、操作系统上使用的字节顺序

CPU	操作系统	字节顺序
x86 (Intel、AMD等)	所有	little-endian
DEC Alpha	所有	little-endian

（续表）

CPU	操作系统	字节顺序
HP-PA	NT	little-endian
HP-PA	UNIX	big-endian
SUN SPARC	所有	big-endian
MIPS	NT	little-endian
MIPS	UNIX	big-endian
PowerPC	NT	little-endian
PowerPC	非NT	big-endian
RS/6000	UNIX	big-endian
Motorola m68k	所有	big-endian

一般来说，我们不用关心字节顺序问题，除非要涉及到跨平台的通信和资源共享，如本章将要介绍的网络编程（网络传输协议TCP/IP采用的是big-endian）。假设现在要在使用不同字节顺序的机器之间传输和交换数据，那该怎么办呢？（同样的数据，不同的机器可能有不同的理解，岂不是有悖初衷！）有两种方法，一种是全部转换成文本来传输，另一种是双方都按照某一方的字节顺序来传输（这时就有一个不同字节顺序之间的相互转换问题）。

Socket编程中经常采用第二种方法。整个传输过程如下：发送端将本机的数据转换成网络的字节顺序（调用API函数htonl或htons），然后发送；接收端收到网络数据后，先将数据转换成本机的字节顺序（调用API函数ntohl或ntohs），然后再进行其他操作——如此就能保证"会议精神"在通信双方的正确传达了！这个过程中用到的几个API函数：ntohl、htonl、ntohs、htons，名字都差不多，很难区分。但是如果知道了它们的来历，问题也就不存在了：n是network，网络的意思；h是host，本地主机的意思。ntohl，就是将32位的u_long类型的数据从网络字节顺序转换成本机字节顺序（htonl的字节顺序转换过程与ntohl相反）；ntohs，就是将16位的u_short类型的数据从网络字节顺序转换成本机字节顺序（htons的字节顺序转换过程与ntohs相反）。

最后还有一个小问题：如何知道本机的字节顺序呢？有个很简单的方法，如下所示：

```
BOOL IsLittleEndian(void)
{
    WORD    wValue = 0x5678;
    return  (*((BYTE*)&wValue) == 0x78);
}
```

另外，TCPServer程序和TCPClient程序在实现时都用到了一个工具类UNetwork。这个类实现了两个静态成员函数：GetHostInfo和DumpSocketError。前者用于获取本地主机的IP地址、机器名等信息，后者用于程序调试时跟踪Socket错误。特别是DumpSocketError函数，非常实用。因为Socket程序的调试一般都比较麻烦，这时DumpSocketError函数就能将整型Socket错误码转换成容易理解的字符串说明形式输出，非常方便！

```
//
// UNetwork.cpp
```

```
//
// 获取本地主机的IP地址和机器名
BOOL UNetwork::GetHostInfo(char * outIP, char * outName)
{
    char  name[300];
    // 获取主机名
    if (gethostname(name, 300) == 0)
    {
        if (outName)
        {
            strcpy(outName, name);
        }
        // 获取主机的IP地址
        PHOSTENT  hostinfo;
        if ((hostinfo = gethostbyname(name)) != NULL)
        {
            LPCSTR pIP = inet_ntoa (*(struct in_addr *)*hostinfo->
                h_addr_list);
            strcpy(outIP, pIP);
            return TRUE;
        }
    }
    return FALSE;
}
// 将整型的Socket错误码转换成字符串说明形式输出
void UNetwork::DumpSocketError(void)
{
    switch (WSAGetLastError())
    {
    case WSANOTINITIALISED:
        TRACE("A successful WSAStartup call must occur before using this
            function. ");
        break;
    case WSAENETDOWN:
        TRACE("The network subsystem has failed. ");
        break;
    case WSAEACCES:
        TRACE("The requested address is a broadcast address, but the
            ppropriate flag was not set. Call setsockopt with the
            SO_BROADCAST parameter to allow the use of the broadcast
            address. ");
        break;
    case WSAEINVAL:
        TRACE("An unknown flag was specified, or MSG_OOB was specified for
            a socket with SO_OOBINLINE enabled. ");
        break;
    case WSAEINTR:
        TRACE("A blocking Windows Sockets 1.1 call was canceled through
            WSACancelBlockingCall. ");
        break;
    case WSAEINPROGRESS:
        TRACE("A blocking Windows Sockets 1.1 call is in progress, or the
```

```
        service provider is still processing a callback function. ");
    break;
case WSAEFAULT:
    TRACE("The buf or to parameters are not part of the user address space,
        or the tolen parameter is too small. ");
    break;
case WSAENETRESET:
    TRACE("The connection has been broken due to keep-alive activity
        detecting a failure while the operation was in progress. ");
    break;
case WSAENOBUFS:
    TRACE("No buffer space is available. ");
    break;
case WSAENOTCONN:
    TRACE("The socket is not connected (connection-oriented sockets
        only). ");
    break;
case WSAENOTSOCK:
    TRACE("The descriptor is not a socket. ");
    break;
case WSAEOPNOTSUPP:
    TRACE("MSG_OOB was specified, but the socket is not stream-style such
        as type SOCK_STREAM, OOB data is not supported in the
        communication domain associated with this socket, or the socket
        is unidirectional and supports only receive operations. ");
    break;
case WSAESHUTDOWN:
    TRACE("The socket has been shut down; it is not possible to sendto
        on a socket after shutdown has been invoked with how set to SD_SEND
        or SD_BOTH. ");
    break;
case WSAEWOULDBLOCK:
    TRACE("The socket is marked as nonblocking and the requested
        operation would block. ");
    break;
case WSAEMSGSIZE:
    TRACE("The socket is message oriented, and the message is larger than
        the maximum supported by the underlying transport. ");
    break;
case WSAEHOSTUNREACH:
    TRACE("The remote host cannot be reached from this host at this time.
        ");
    break;
case WSAECONNABORTED:
    TRACE("The virtual circuit was terminated due to a time-out or other
        failure. The application should close the socket as it is no
        longer usable. ");
    break;
case WSAECONNRESET:
    TRACE("The virtual circuit was reset by the remote side executing
        a hard or abortive close. For UPD sockets, the remote host was
        unable to deliver a previously sent UDP datagram and responded
```

```
            with a \"Port Unreachable\" ICMP packet. The application should
            close the socket as it is no longer usable. ");
        break;
    case WSAEADDRNOTAVAIL:
        TRACE("The remote address is not a valid address, for example,
            ADDR_ANY. ");
        break;
    case WSAEAFNOSUPPORT:
        TRACE("Addresses in the specified family cannot be used with this
            socket. ");
        break;
    case WSAEDESTADDRREQ:
        TRACE("A destination address is required. ");
        break;
    case WSAENETUNREACH:
        TRACE("The network cannot be reached from this host at this time.
            ");
        break;
    case WSAETIMEDOUT:
        TRACE("The connection has been dropped, because of a network failure
            or because the system on the other end went down without notice.
            ");
        break;
    default:
        TRACE("Unknown socket error. ");
        break;
    }
}
```

提示：本书配套光盘的SourceCodes\Chapter03\TCPDemo目录下提供了TCP传输演示程序的完整实现。其中，TCPServer为服务器程序，TCPClient为客户机程序。打开WsClientServer.dsw可以同时浏览TCPServer和TCPClient两个项目。

3.1.2 UDP传输

UDP（User Datagram Protocol，用户数据报协议）是一种无连接的网络传输协议。UDP协议提供的是一种基本的、低延时的、称为数据报的传输服务。UDP传输不像TCP传输一样需要预先建立一条连接；UDP没有计时机制、流控或拥塞管理机制，由于某种原因造成丢失的数据报也不会被重传。因此，UDP提供的是一种不可靠的应用数据传输服务。

提示：TCP与UDP之间的主要差别在于可靠性。但也不是说，因为UDP是一种不可靠的传输协议而一无用处。在一个良好的网络环境下（比如局域网内），使用UDP传输数据还是相当可靠的，而且效率比较高。UDP（比TCP）更适合于与时间相关的应用数据的传输。

本节将要实现一个UDP传输的演示程序UDPDemo，它包括服务器和客户机两个部分。它们的程序界面如图3.8所示。

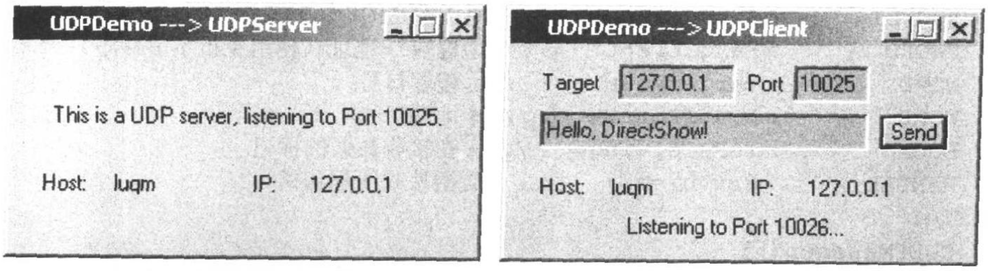


图 3.8　UDP 传输演示程序界面

UDPDemo的演示过程如下：

（1）将服务器和客户机两部分程序都运行起来（此时服务器启动了用于接收10025端口数据的子线程，而客户机也启动了用于接收10026端口数据的子线程）。

（2）在客户机程序界面上输入服务器的IP地址（如果服务器和客户机运行在同一台机器上，IP地址可以指定为127.0.0.1）、数据接收的端口号（因为服务器在10025端口上接收数据，这里也应该指定为10025）。

（3）在客户机程序界面上编辑欲发送消息的内容，然后单击Send按钮向服务器发出UDP数据。

（4）服务器接收到客户机发送过来的数据后就弹出如图3.9左图所示的消息框。随后，服务器向客户机发送一个反馈消息。

（5）客户机接收到服务器发送过来的反馈数据后也弹出一个消息框，如图3.9的右图所示。

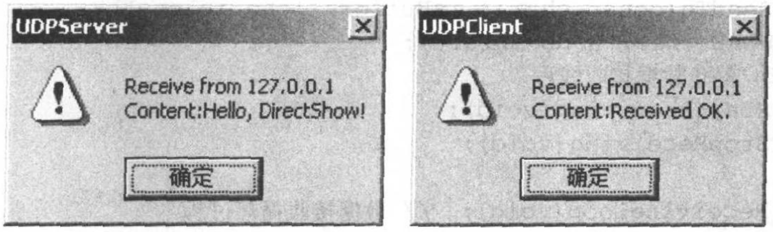


图 3.9　UDP 传输过程中显示的一对消息框

UDPDemo在具体实现时，设计了一个CUDPManager类专门负责UDP数据的发送和接收（数据接收使用一个独立的子线程）。CUDPManager类的定义和实现如下：

```
//
// CUDPManager.h
//

#ifndef __H_CUDPManager__
#define __H_CUDPManager__

class CUDPManager
{
private:
    SOCKET      mSckReceiver;    // 用于接收的Socket
```

```
    SOCKET      mSckSender;      // 用于发送的Socket
    DWORD       mTargetIP;       // 远程端IP地址（使用主机字节顺序）
    WORD        mTargetPort;     // 远程端口号
    WORD        mLocalPort;      // 本地端口号
    BOOL        mIsReceiving;    // 正在接收数据的标记
    HANDLE      mRcvThread;      // 数据接收线程句柄
public:
    CUDPManager();
    ~CUDPManager();
    // 设置/获取远程端的IP地址
    void SetTargetIP(DWORD inIP);
    DWORD GetTargetIP(void);
    void SetTargetIP(const char * inIP);
    void GetTargetIP(char * outIP);
    // 设置/获取远程端口号
    void SetTargetPort(WORD inPort);
    WORD GetTargetPort(void);
    // 设置/获取本地端口号
    void SetLocalPort(WORD inPort);
    WORD GetLocalPort(void);
    // 创建/销毁用于发送的Socket
    BOOL CreateSender(void);
    void DeleteSender(void);
    // 创建/销毁用于接收的Socket
    BOOL CreateReceiver(void);
    void DeleteReceiver(void);
    // 使用UDP协议发送数据的两个函数
    BOOL Send(const char * inBuffer, long inLength);
    BOOL SendTo(const char * inBuffer, long inLength,
        DWORD inIP, WORD inPort);
    // 启动/停止数据接收线程
    BOOL StartReceiving(void);
    void StopReceiving(void);
private:
    void ReceivingLoop(void);  // 数据接收循环过程
    static DWORD WINAPI ReceivingThrd(void * pParam);  // 接收线程执行体
};

#endif // __H_CUDPManager__

//
// CUDPManager.cpp
//

#include "stdafx.h"
#include "CUDPManager.h"
#include "UNetwork.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
```

```
#endif

//////////////////////////////////////////////////////////////////////////
CUDPManager::CUDPManager()
{
    // 参数初始化
    mSckReceiver    = INVALID_SOCKET;
    mSckSender      = INVALID_SOCKET;
    mTargetIP       = 0x7f000001; // 127.0.0.1
    mTargetPort     = 10080;
    mLocalPort      = 10080;
    mIsReceiving    = FALSE;
    mRcvThread      = NULL;
}
CUDPManager::~CUDPManager()
{
    // 销毁所有使用过的Socket
    DeleteSender();
    DeleteReceiver();
    StopReceiving();
}
// 设置远程端的IP地址，参数是DWORD类型
void CUDPManager::SetTargetIP(DWORD inIP)
{
    mTargetIP = inIP;
}
// 得到远程端的IP地址
DWORD CUDPManager::GetTargetIP(void)
{
    return mTargetIP;
}
// 重载函数：设置远程端的IP地址，参数是字符串类型
void CUDPManager::SetTargetIP(const char * inIP)
{
    // 将IP地址从字符串形式转换成DWORD类型（使用主机字节顺序）
    mTargetIP = ntohl(inet_addr(inIP));
}
// 重载函数：得到远程端的字符串形式的IP地址
void CUDPManager::GetTargetIP(char * outIP)
{
    if (outIP)
    {
        // 将IP地址从DWORD类型转换成字符串形式
        struct in_addr   in;
        in.S_un.S_addr = htonl(mTargetIP);
        char * pStr = inet_ntoa(in);
        strcpy(outIP, pStr);
    }
}
// 设置远程端口号
void CUDPManager::SetTargetPort(WORD inPort)
{
```

```
        mTargetPort = inPort;
    }
    // 得到远程端口号
    WORD CUDPManager::GetTargetPort(void)
    {
        return mTargetPort;
    }
    // 设置本地端口号
    void CUDPManager::SetLocalPort(WORD inPort)
    {
        mLocalPort = inPort;
    }
    // 得到本地端口号
    WORD CUDPManager::GetLocalPort(void)
    {
        return mLocalPort;
    }
    // 创建用于发送的Socket
    BOOL CUDPManager::CreateSender(void)
    {
        DeleteSender();
        // 创建一个UDP传输的Socket
        mSckSender = socket(AF_INET, SOCK_DGRAM, 0);
        if (mSckSender != INVALID_SOCKET)
        {
            return TRUE;
        }
        return FALSE;
    }
    // 销毁用于发送的Socket
    void CUDPManager::DeleteSender(void)
    {
        if (mSckSender != INVALID_SOCKET)
        {
            closesocket(mSckSender);
            mSckSender = INVALID_SOCKET;
        }
    }
    // 创建用于接收的Socket
    BOOL CUDPManager::CreateReceiver(void)
    {
        DeleteReceiver();
        // 创建一个UDP传输的Socket
        mSckReceiver = socket(AF_INET, SOCK_DGRAM, 0);
        if (mSckReceiver != INVALID_SOCKET)
        {
            // 在Socket上设置参数：允许地址复用
            BOOL flag = TRUE;
            int ret = setsockopt(mSckReceiver, SOL_SOCKET, SO_REUSEADDR,
                (char *) &flag, sizeof(flag));
            if (ret == SOCKET_ERROR)
            {
```

```
            DeleteReceiver();
            return FALSE;
        }
        // 将Socket绑定到本地端口号上
        SOCKADDR_IN  addr;
        addr.sin_family         = AF_INET;
        addr.sin_addr.s_addr    = htonl(INADDR_ANY);
        addr.sin_port           = htons(mLocalPort);
        ret = bind(mSckReceiver, (struct sockaddr*) &addr, sizeof(addr));
        if (ret == SOCKET_ERROR)
        {
            DeleteReceiver();
            return FALSE;
        }
        return TRUE;
    }
    return FALSE;
}
// 销毁用于接收的Socket
void CUDPManager::DeleteReceiver(void)
{
    if (mSckReceiver != INVALID_SOCKET)
    {
        closesocket(mSckReceiver);
        mSckReceiver = INVALID_SOCKET;
    }
}
// 使用已经创建好的用于发送的Socket发送数据
BOOL CUDPManager::Send(const char * inBuffer, long inLength)
{
    SOCKADDR_IN  addr;
    memset((char *) &addr, 0, sizeof(addr));
    addr.sin_family      = AF_INET;
    addr.sin_addr.s_addr = htonl(mTargetIP);
    addr.sin_port        = htons(mTargetPort);
    // 使用Socket发送数据
    int val = sendto(mSckSender, inBuffer, inLength, 0,
        (sockaddr *) &addr, sizeof(addr));
    return (val != SOCKET_ERROR);
}
// 创建一个新的Socket，并用它将数据发送到指定的IP地址、端口号上
BOOL CUDPManager::SendTo(const char * inBuffer, long inLength, DWORD inIP,
    WORD inPort)
{
    // 创建一个UDP传输的Socket
    SOCKET sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock != INVALID_SOCKET)
    {
        SOCKADDR_IN  addr;
        memset((char *) &addr, 0, sizeof(addr));
        addr.sin_family         = AF_INET;
        addr.sin_addr.s_addr    = htonl(inIP);
```

```
        addr.sin_port           = htons(inPort);
        // 发送数据
        int val = sendto(sock, inBuffer, inLength, 0,
            (sockaddr *) &addr, sizeof(addr));
        if (val == SOCKET_ERROR)
        {
            // 跟踪Socket错误
#ifdef _DEBUG
            UNetwork::DumpSocketError();
#endif
        }
        // 发送完成后销毁该Socket
        closesocket(sock);
        return (val != SOCKET_ERROR);
    }
    return FALSE;
}
// 启动数据接收线程（因为调用recvfrom函数接收UDP数据时会阻塞）
BOOL CUDPManager::StartReceiving(void)
{
    if (mSckReceiver == INVALID_SOCKET)
    {
        CreateReceiver();
    }

    if (mSckReceiver != INVALID_SOCKET)
    {
        if (mIsReceiving)
        {
            return TRUE;
        }
        DWORD threadID = 0;
        mRcvThread = CreateThread(NULL, 0, ReceivingThrd,
            this, 0, &threadID);
        return (mRcvThread != NULL);
    }
    return FALSE;
}
// 停止数据接收线程
void CUDPManager::StopReceiving(void)
{
    if (mIsReceiving)
    {
        // 销毁Socket以使接收函数失败或脱离阻塞
        DeleteReceiver();
        // 等待接收线程完全退出
        if (mRcvThread != NULL)
        {
            WaitForSingleObject(mRcvThread, INFINITE);
            mRcvThread = NULL;
        }
    }
```

```
}
// 线程函数执行体：调用本类的ReceivingLoop函数
DWORD WINAPI CUDPManager::ReceivingThrd(void * pParam)
{
    ASSERT(pParam);
    CUDPManager * pController = (CUDPManager*) pParam;
    pController->ReceivingLoop();
    return 0;
}
// 数据接收过程
void CUDPManager::ReceivingLoop(void)
{
    struct sockaddr_in  addr_cli;
    int  addr_cli_len = sizeof(addr_cli);
    char buffer[1024];  // 发送数据缓存
    long bytes = 0;

    mIsReceiving = TRUE;
    while (mIsReceiving)
    {
        // 等待接收数据
        int addr_cli_len = sizeof(addr_cli);
        bytes = recvfrom(mSckReceiver, (char *)buffer, 1024, 0,
            (LPSOCKADDR) &addr_cli, (int *) &addr_cli_len);
        if (bytes == SOCKET_ERROR || bytes == 0)
        {
            // 如果Socket发送错误或者Socket断开，则跳出循环
            mIsReceiving = FALSE;
        }
        else
        {
            buffer[bytes] = '\0';
            // 获取远程端的IP地址
            char * pStr  = inet_ntoa(addr_cli.sin_addr);
            // 检查标记：是否需要发出反馈消息
            // 作为演示，发送的UDP数据包第一个字节用于指示是否需要反馈：1表示需要反
            // 馈，0表示不需要反馈
            if (buffer[0] == '1')
            {
                // 向远程端发出一个反馈消息，第一个字节指定为0，表示不再需要远程端反
                // 馈，否则通信双方的数据传输永无休止！
                CString str = "0Received OK." ;
                SendTo(str,str.GetLength()+1,ntohl(inet_addr(pStr)),
                    10026);
            }
            // 弹出一个对话框显示接收到的数据内容
            CString msg;
            msg.Format("Receive from %s \nContent:%s", pStr, buffer+1);
            AfxMessageBox(msg);
        }
    }
}
```

再来看服务器程序UDPServer和客户机程序UDPClient的具体实现。它们都是基于对话框的MFC程序，并且分别在各自的主对话框类中定义了一个CUDPManager类的实例。然后就是在对话框的初始化函数中进行一系列的操作，包括创建UDP传输用的Socket、启动数据接收线程等。

```
// UDPServerDlg.h : header file
//
class CUDPServerDlg : public CDialog
{
protected:
    CUDPManager  mUDPManager;
    // 其他成员定义（省略）
    // ……
};

// UDPServerDlg.cpp : implementation file
//
// 服务器程序主对话框的初始化
BOOL CUDPServerDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // Set the icon for this dialog.  The framework does this automatically
    //  when the application's main window is not a dialog
    SetIcon(m_hIcon, TRUE);          // Set big icon
    SetIcon(m_hIcon, FALSE);         // Set small icon

    // 获取本地机器的IP地址、机器名，并在界面上显示
    char hostName[100];
    char hostIP[50];
    if (UNetwork::GetHostInfo(hostIP, hostName))
    {
        mEditHostName.SetWindowText(hostName);
        mEditHostIP.SetWindowText(hostIP);
    }
    // 服务器在10025端口上接收数据
    mServerInfo.Format("This is a UDP server, listening to Port 10025.");
    UpdateData(FALSE);
    // 设置服务器接收数据用的端口号，然后创建接收用的Socket
    mUDPManager.SetLocalPort(10025);
    if (mUDPManager.CreateReceiver())
    {
        // 启动数据接收线程
        mUDPManager.StartReceiving();
    }

    return TRUE;  // return TRUE  unless you set the focus to a control
}

// UDPClientDlg.h : header file
//
```

```
class CUDPClientDlg : public CDialog
{
protected:
    CUDPManager  mUDPManager;
    // 其他成员定义（省略）
    // ……
};

// UDPClientDlg.cpp : implementation file
//
// 客户机程序主对话框的初始化
BOOL CUDPClientDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // Set the icon for this dialog.  The framework does this automatically
    //  when the application's main window is not a dialog
    SetIcon(m_hIcon, TRUE);          // Set big icon
    SetIcon(m_hIcon, FALSE);         // Set small icon

    // 获取本地机器的IP地址、机器名，并在界面上显示
    char hostName[100];
    char hostIP[50];
    if (UNetwork::GetHostInfo(hostIP, hostName))
    {
        mEditHostName.SetWindowText(hostName);
        mEditHostIP.SetWindowText(hostIP);
    }
    // 客户机在10026端口上接收数据
    mUDPManager.SetLocalPort(10026);
    // 创建接收用的Socket
    if (mUDPManager.CreateReceiver())
    {
        // 启动数据接收线程
        mUDPManager.StartReceiving();
    }
    // 创建发送用的Socket
    mUDPManager.CreateSender();

    return TRUE;  // return TRUE  unless you set the focus to a control
}
// 客户机界面上的Send按钮的响应函数
void CUDPClientDlg::OnButtonSend()
{
    UpdateData(TRUE);
    // 设置远程端的IP地址、数据接收端口号
    mUDPManager.SetTargetIP(mTargetIP);
    mUDPManager.SetTargetPort((WORD)mTargetPort);
    // 向远程端发送数据
    // 发送数据的第一个字节指定为1，表示要求远程端接收到数据后发回一个反馈消息
    mUDPManager.Send("1" + mMsg2Send, mMsg2Send.GetLength() + 1);
}
```

提示：本书配套光盘的SourceCodes\Chapter03\UDPDemo目录下提供了UDP传输演示程序的完整实现。其中，UDPServer为服务器程序，UDPClient为客户机程序。打开WsClientServer.dsw可以同时浏览UDPServer和UDPClient两个项目。

3.1.3 IP组播技术

组播技术被认为是WWW技术推广之后出现的最激动人心的网络技术之一。组播是一种允许一个或多个发送者发送单一的数据包到多个接收者的网络技术。组播源把数据包发送到特定的组播组（Multicast Group），而只有加入到该组播组的主机才能接收到这些数据包。组播可以大大节省网络带宽，因为无论有多少个目标地址，在整个网络的任何一条链路上只传送单一的数据包。单播与组播数据传送过程的区别如图3.10所示。

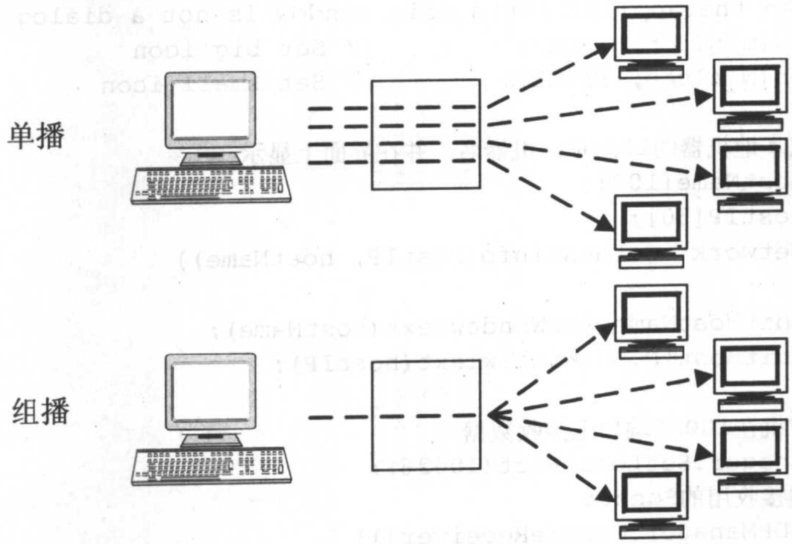


图 3.10　单播与组播的数据传送过程

单播（Unicast）传输：在发送者和每一接收者之间实现点对点的网络连接。如果一个发送者同时给多个接收者传输相同的数据，则必须相应地将数据包复制成多份后再分别投递。如果有大量主机希望获得数据包的同一份拷贝，将导致发送者负担沉重、延时长、网络拥塞，为保证一定的服务质量需增加硬件和带宽。

组播（Multicast）传输：在发送者和每一接收者之间实现一点对多点的网络连接。如果一个发送者同时给多个接收者传输相同的数据，只需投递一份数据包就可以了。组播提高了数据的传送效率，减少了骨干网络出现拥塞的可能性。

广播（Broadcast）传输：是指在IP子网内广播数据包，所有在子网内部的主机都将收到这些数据包，不管它们是否乐于接收。广播的使用范围非常小，只在本地子网内有效，因为路由器通常会封锁广播通信。广播传输会增加非接收者的开销。

目前，使用得最为广泛的组播技术是IP组播技术。IP组播技术是一种为优化使用网络资源而产生的技术，通常用于多点工作方式下的应用程序中，它是标准IP网络层协议技术的一个扩展。从Steve Deering于1989年提出的RFC 1112（Host Extensions for IP Multicasting）

中的定义可以得知，IP组播的核心思想是：通过一个IP地址向一组主机发送数据（UDP包）；发送者仅仅向一个组地址发送信息，接收者只需加入到这个分组就可以接收信息；所有的接收者接收的是同一个数据流；组中成员是动态的，可以根据自己的意愿随时随意加入或退出；每一台主机都可以同时加入到多个组中，每一个组播地址可以在不同的端口或者不同的Socket上有多个数据流，同时许多实际应用可以共享一个组地址。IP组播技术可以有效地避免重复发送可能引起的广播风暴，并且能够突破路由器的限制，将数据包传送到其他网段。

IP地址方案专门为组播划出了一个地址范围，在IPv4中为D类地址，范围是224.0.0.0~239.255.255.255，并将D类地址划分为局部链接组播地址、预留组播地址、管理权限组播地址等，分配如下：

- 局部链接地址：224.0.0.0～224.0.0.255，用于局域网，路由器不转发此范围内的IP包。
- 预留组播地址：224.0.1.0～238.255.255.255，用于全球范围或网络协议。
- 管理权限地址：239.0.0.0～239.255.255.255，组织内部使用，用于限制组播范围。

从多媒体应用的角度来看，IP组播技术的使用对于网络视频的多点实时传输、网络多点实时监控具有特别重要的意义。本节将实现一个IP组播的演示程序MulticastDemo，它包括服务器和客户机两个部分。它们的程序界面如图3.11所示。

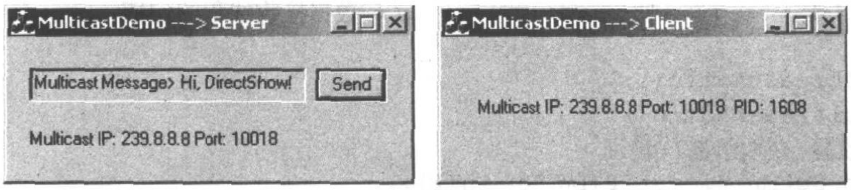


图3.11 组播传输演示程序界面

MulticastDemo的演示过程如下：

（1）将服务器和客户机两部分程序都运行起来。为了演示组播特性，客户机程序可以运行多个进程，各个进程以PID来区别；它们都加入到同一个组播组239.8.8.8，并且各自启动了一个数据接收线程。

（2）在服务器程序界面上编辑欲发送的消息内容，然后单击Send按钮发送数据。

（3）所有运行起来的客户机程序进程都将接收到服务器发出的信息，并且弹出如图3.12所示的消息框。（假设运行了两个客户机程序进程，它们的PID分别是1608和1716。）

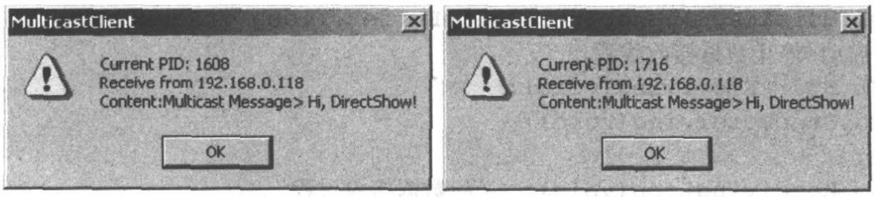


图3.12 组播传输客户机进程接收到数据后显示的消息框

提示：进行组播演示之前，请确认你的机器安装有网卡，并且局域网连接正常。

组播实际上使用了UDP协议进行数据的传输。MulticastDemo在具体实现时，设计了一个CMulticastAdmin类专门负责组播数据的发送和接收（数据接收使用一个独立的子线程）。（CMulticastAdmin类实际上是由CUDPManager 类改写过来的，这两个类的实现非常类似。）CMulticastAdmin类的定义和实现如下：

```
//
// CMulticastAdmin.h
//

#ifndef __H_CMulticastAdmin__
#define __H_CMulticastAdmin__

class CMulticastAdmin
{
private:
    SOCKET      mSckReceiver;       // 接收用的Socket
    SOCKET      mMulticaster;       // 组播发送用的Socket
    DWORD       mMulticastIP;       // 组播IP地址（使用主机字节顺序）
    WORD        mMulticastPort;     // 组播的端口号
    BOOL        mIsReceiving;       // 数据正在接收的标记
    HANDLE      mRcvThread;         // 数据接收线程的句柄
public:
    CMulticastAdmin();
    ~CMulticastAdmin();
    // 设置/获取组播IP地址
    void SetMulticastIP(DWORD inIP);
    DWORD GetMulticastIP(void);
    void SetMulticastIP(const char * inIP);
    void GetMulticastIP(char * outIP);
    // 设置/获取组播端口号
    void SetMulticastPort(WORD inPort);
    WORD GetMulticastPort(void);
    // 创建/销毁组播用的Socket
    BOOL CreateMulticaster(void);
    void DeleteMulticaster(void);
    // 创建/销毁接收用的Socket
    BOOL CreateReceiver(void);
    void DeleteReceiver(void);
    // （组播）发送数据
    BOOL Multicast(const char * inBuffer, long inLength);
    // 启动/停止数据接收线程
    BOOL StartReceiving(void);
    void StopReceiving(void);
private:
    void ReceivingLoop(void);  // 数据接收过程
    static DWORD WINAPI ReceivingThrd(void * pParam);  // 接收线程执行体
};
```

```
#endif // __H_CMulticastAdmin__

//
// CMulticastAdmin.cpp
//

#include "stdafx.h"
#include "CMulticastAdmin.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//////////////////////////////////////////////////////////////////////////
CMulticastAdmin::CMulticastAdmin()
{
    // 参数初始化
    mSckReceiver    = INVALID_SOCKET;
    mMulticaster    = INVALID_SOCKET;
    // Multicast IP: from 224.0.0.0 to 239.255.255.255
    mMulticastIP    = 0xef080808; // 239.8.8.8
    mMulticastPort  = 10018;
    mIsReceiving    = FALSE;
    mRcvThread      = NULL;
}
CMulticastAdmin::~CMulticastAdmin()
{
    // 销毁所有使用过的Socket
    DeleteMulticaster();
    DeleteReceiver();
    StopReceiving();
}
// 设置组播IP地址
void CMulticastAdmin::SetMulticastIP(DWORD inIP)
{
    mMulticastIP = inIP;
}
// 获取组播IP地址
DWORD CMulticastAdmin::GetMulticastIP(void)
{
    return mMulticastIP;
}
// 设置组播IP地址
void CMulticastAdmin::SetMulticastIP(const char * inIP)
{
    mMulticastIP = ntohl(inet_addr(inIP));
}
// 获取组播IP地址
void CMulticastAdmin::GetMulticastIP(char * outIP)
{
```

```
        if (outIP)
        {
            struct in_addr   in;
            in.S_un.S_addr = htonl(mMulticastIP);
            char * pStr = inet_ntoa(in);
            strcpy(outIP, pStr);
        }
}
// 设置组播端口号
void CMulticastAdmin::SetMulticastPort(WORD inPort)
{
    mMulticastPort = inPort;
}
// 获取组播端口号
WORD CMulticastAdmin::GetMulticastPort(void)
{
    return mMulticastPort;
}
// 创建组播发送用的Socket
BOOL CMulticastAdmin::CreateMulticaster(void)
{
    DeleteMulticaster();
    // 初始化Socket函数库：使用2.2版本的WinSock DLL
    WSADATA  data;
    int ret = WSAStartup(0x0202, &data);
    if (ret != 0)
    {
        WSACleanup();
        return FALSE;
    }
    // 创建一个使用UDP协议传输的Socket
    mMulticaster = socket(AF_INET, SOCK_DGRAM, 0);
    if (mMulticaster == INVALID_SOCKET)
    {
        WSACleanup();
        return FALSE;
    }
    // 在Socket上设置参数：允许地址复用
    BOOL flag = TRUE;
    ret = setsockopt(mMulticaster, SOL_SOCKET, SO_REUSEADDR,
        (char *)&flag, sizeof(flag));
    // 绑定Socket到组播端口号上
    SOCKADDR_IN addr;
    ZeroMemory(&addr, sizeof(addr));
    addr.sin_family      = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;   // 不关心网卡地址
    addr.sin_port        = htons(mMulticastPort);
    ret = bind(mMulticaster, (struct sockaddr*) &addr, sizeof(addr));
    return TRUE;
}
// 销毁组播发送用的Socket
void CMulticastAdmin::DeleteMulticaster(void)
```

```
{
    if (mMulticaster != INVALID_SOCKET)
    {
        closesocket(mMulticaster);
        mMulticaster = INVALID_SOCKET;
        WSACleanup();
    }
}
// 创建接收用的Socket（需要加入指定的组播组）
BOOL CMulticastAdmin::CreateReceiver(void)
{
    DeleteReceiver();
    // 初始化Socket函数库：使用2.2版本的WinSock DLL
    WSADATA  data;
    int ret = WSAStartup(0x0202, &data);
    if (ret != 0)
    {
        WSACleanup();
        return FALSE;
    }
    // 创建一个使用UDP协议传输的Socket
    mSckReceiver = socket(AF_INET, SOCK_DGRAM, 0);
    if (mSckReceiver == INVALID_SOCKET)
    {
        WSACleanup();
        return FALSE;
    }
    // 在Socket上设置参数：允许地址复用
    BOOL flag = TRUE;
    ret = setsockopt(mSckReceiver, SOL_SOCKET, SO_REUSEADDR,
        (char *) &flag, sizeof(flag));
    // 绑定Socket到组播端口号上
    SOCKADDR_IN addr;
    ZeroMemory(&addr, sizeof(addr));
    addr.sin_family     = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;   // 不关心网卡地址
    addr.sin_port        = htons(mMulticastPort);
    ret = bind(mSckReceiver, (struct sockaddr*) &addr, sizeof(addr));
    // 将Socket加入到组播组（以便接收组播数据）
    struct ip_mreq mreq;
    mreq.imr_multiaddr.s_addr = htonl(mMulticastIP);
    mreq.imr_interface.s_addr = INADDR_ANY;
    ret = setsockopt(mSckReceiver, IPPROTO_IP, IP_ADD_MEMBERSHIP,
        (char *) &mreq, sizeof(mreq));
    return TRUE;
}
// 销毁接收用的Socket
void CMulticastAdmin::DeleteReceiver(void)
{
    if (mSckReceiver != INVALID_SOCKET)
    {
        closesocket(mSckReceiver);
```

```
            mSckReceiver = INVALID_SOCKET;
            WSACleanup();
        }
}
// 组播发送数据
BOOL CMulticastAdmin::Multicast(const char * inBuffer, long inLength)
{
    SOCKADDR_IN  addr;
    memset((char *) &addr, 0, sizeof(addr));
    addr.sin_family      = AF_INET;
    addr.sin_addr.s_addr = htonl(mMulticastIP);
    addr.sin_port        = htons(mMulticastPort);
    // 向组播IP地址、端口号发送数据
    int val = sendto(mMulticaster, inBuffer, inLength, 0,
        (sockaddr *) &addr, sizeof(addr));
    return (val != SOCKET_ERROR);
}
// 启动数据接收线程
BOOL CMulticastAdmin::StartReceiving(void)
{
    if (mSckReceiver == INVALID_SOCKET)
    {
        CreateReceiver();
    }

    if (mSckReceiver != INVALID_SOCKET)
    {
        if (mIsReceiving)
        {
            return TRUE;
        }
        DWORD threadID = 0;
        mRcvThread = CreateThread(NULL, 0, ReceivingThrd,
            this, 0, &threadID);
        return (mRcvThread != NULL);
    }
    return FALSE;
}
// 停止数据接收线程
void CMulticastAdmin::StopReceiving(void)
{
    if (mIsReceiving)
    {
        // 销毁接收用的Socket，使接收函数失败或脱离阻塞
        DeleteReceiver();
        // 等待接收线程完全退出
        if (mRcvThread != NULL)
        {
            WaitForSingleObject(mRcvThread, INFINITE);
            mRcvThread = NULL;
        }
    }
```

```
}
// 数据接收线程的执行体（实际调用本类的ReceivingLoop函数）
DWORD WINAPI CMulticastAdmin::ReceivingThrd(void * pParam)
{
    ASSERT(pParam);
    CMulticastAdmin * pController = (CMulticastAdmin*) pParam;
    pController->ReceivingLoop();
    return 0;
}
// 数据接收过程
void CMulticastAdmin::ReceivingLoop(void)
{
    struct sockaddr_in  addr_cli;
    int  addr_cli_len = sizeof(addr_cli);
    char buffer[1024];  // 数据接收缓存
    long bytes = 0;

    mIsReceiving = TRUE;
    while (mIsReceiving)
    {
        // 等待接收数据
        int addr_cli_len = sizeof(addr_cli);
        bytes = recvfrom(mSckReceiver, (char *)buffer, 1024, 0,
            (LPSOCKADDR) &addr_cli, (int *) &addr_cli_len);
        if (bytes == SOCKET_ERROR || bytes == 0)
        {
            mIsReceiving = FALSE;
        }
        else
        {
            // 获取发送者的IP地址
            buffer[bytes] = '\0';
            char *  pStr  = inet_ntoa(addr_cli.sin_addr);
            // 弹出一个消息框，显示收到的数据内容
            CString msg;
            msg.Format("Current PID: %d\nReceive from %s \nContent:%s",
                GetCurrentProcessId(), pStr, buffer);
            AfxMessageBox(msg);
        }
    }
}
```

再来看服务器程序MulticastServer和客户机程序MulticastClient的具体实现。它们都是基于对话框的MFC程序，并且分别在各自的主对话框类中定义了一个CMulticastAdmin类的实例。然后就是在对话框的初始化函数中进行一系列的操作，包括创建组播发送/接收用的Socket、启动数据接收线程等。

```
// MulticastServerDlg.h : header file
//
class CMulticastServerDlg : public CDialog
{
```

```
protected:
    CMulticastAdmin     mMulticaster;
    // 其他成员定义（省略）
    // ……
};

// MulticastServerDlg.cpp : implementation file
//
CMulticastServerDlg::CMulticastServerDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CMulticastServerDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CMulticastServerDlg)
    mMsg = _T("Multicast Message> Hi, DirectShow!");
    //}}AFX_DATA_INIT
    // Note that LoadIcon does not require a subsequent DestroyIcon in Win32
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}
// 组播服务器程序主对话框的初始化
BOOL CMulticastServerDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // Set the icon for this dialog.  The framework does this automatically
    //  when the application's main window is not a dialog
    SetIcon(m_hIcon, TRUE);         // Set big icon
    SetIcon(m_hIcon, FALSE);        // Set small icon

    // 在界面上显示组播组IP地址、端口号等信息
    mStaticInfo.SetWindowText("Multicast IP: 239.8.8.8 Port: 10018");
    mMulticaster.SetMulticastIP("239.8.8.8");
    // 创建组播发送用的Socket
    mMulticaster.CreateMulticaster();

    return TRUE;  // return TRUE  unless you set the focus to a control
}
// 服务器程序界面上的Send按钮的响应函数
void CMulticastServerDlg::OnButtonSend()
{
    UpdateData();
    // 向组播组发送一个消息
    mMulticaster.Multicast(mMsg, mMsg.GetLength());
}

// MulticastClientDlg.h : header file
//
class CMulticastClientDlg : public CDialog
{
protected:
    CMulticastAdmin     mReceiver;
    // 其他成员定义（省略）
```

```
    // ......
};

// MulticastClientDlg.cpp : implementation file
//
// 客户机程序主对话框的初始化
BOOL CMulticastClientDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // Set the icon for this dialog.  The framework does this automatically
    //  when the application's main window is not a dialog
    SetIcon(m_hIcon, TRUE);         // Set big icon
    SetIcon(m_hIcon, FALSE);        // Set small icon

    // 在界面上显示组播组IP地址、端口号等信息
    CString info;
    info.Format("Multicast IP: 239.8.8.8 Port: 10018  PID: %d",
        GetCurrentProcessId());
    mStaticInfo.SetWindowText(info);
    // 设置组播组IP地址，创建接收用的Socket
    mReceiver.SetMulticastIP("239.8.8.8");
    if (mReceiver.CreateReceiver())
    {
        // 启动数据接收线程
        mReceiver.StartReceiving();
    }

    return TRUE;  // return TRUE  unless you set the focus to a control
}
```

提示：本书配套光盘的SourceCodes\Chapter03\MulticastDemo目录下提供了组播传输演示程序的完整实现。其中，MulticastServer为服务器程序，MulticastClient为客户机程序。打开WsClientServer.dsw可以同时浏览MulticastServer和MulticastClient两个项目。

3.2 视频聊天

3.2.1 功能介绍

运用DirectShow技术可以进行音视频采集（获得多媒体数据），运用网络技术可以实现主机之间的通信（数据的双向传输），那么如果将这两项技术结合起来，是不是很容易就能开发出视频（兼带语音）的聊天软件呢？答案是肯定的。

本节将要实现一个点对点视频聊天的演示程序AVChat，它的界面流程如图3.13所示。

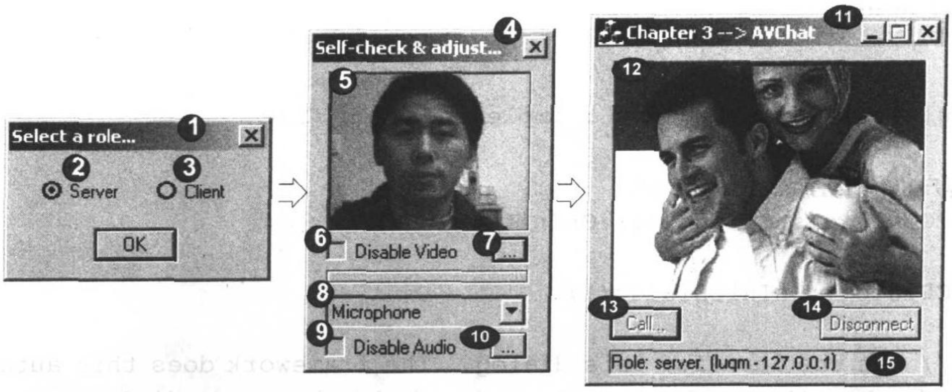


图 3.13　AVChat 的界面流程

① 角色选择对话框

聊天双方必须一个作为服务器，一个作为客户机。服务器是被动的，只有等待客户机的呼叫才能建立连接。AVChat程序将服务器功能和客户机功能合并在一起实现，因此程序启动后首先要使用这个对话框进行角色分配。

② 服务器角色单选按钮

③ 客户机角色单选按钮

④ 本地采集设备自检、调整对话框

在开始音视频采集、网络传输之前，进行本地采集设备相关的一些调整。如视频采集方面，是否检测到有摄像头、摄像头是否对准了目标、图像显示是否正常、是否允许将本地采集的视频发送给远程端等；再如音频采集方面，是否检测到有音频采集设备、音频输入端子是否正确、麦克风连接是否正常、是否允许将本地采集的音频发送给远程端等。

⑤ 本地视频预览窗口

可以通过本窗口中显示的图像判断摄像头是否对准了目标。如果本地机器没有检测到视频采集设备，则预览窗口中无图像显示。

⑥ 是否将本地视频传送给远程端的复选框

虽然本地可以进行正常的视频采集，但如果用户不想让本地视频传送到远程端，则可以将其选中。

⑦ 用于显示本地视频采集参数调整对话框的按钮

单击此按钮可以显示视频采集设备Filter的属性页。用户可以在该属性页上进行一系列视频采集参数的调整。

⑧ 音频输入端子

一般情况下，AVChat程序会自动选择麦克风输入。如果程序自动选择有误，用户应当自行调整。如果麦克风输入正常，用户能够听到自己的说话声（通过麦克风输入，经过采集后送到声卡播放）。

⑨ 是否将本地音频传送给远程端的复选框

虽然本地的音频采集设备能够正常工作，但如果用户手头没有麦克风或者只是不想让本地音频传送到远程端，则可以将其选中。

⑩ 用于显示本地音频采集参数调整对话框的按钮

单击此按钮可以显示音频采集设备Filter的属性页。用户可以在该属性页上进行一系列音频采集参数的调整。

⑪ AVChat程序主界面

⑫ 远程端传送过来的视频图像显示窗口

⑬ 呼叫服务器的按钮

此按钮仅在AVChat作为客户机程序时可用，单击后将弹出如图3.14所示的对话框：输入服务器的IP地址后单击OK按钮，即可向该服务器发出连接请求（要求进行视频通话）。

⑭ 断开连接

服务器和客户机任何一方想要中止当前的视频通话均可单击此按钮。

⑮ 信息提示栏

显示一些提示信息，如本程序的角色（服务器或客户机）、本机的IP地址等。

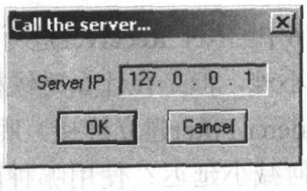


图 3.14 指定服务器的 IP 地址

3.2.2 实现原理

从DirectShow的角度来看，AVChat程序只需实现类似于图3.15和图3.16的Filter Graph就行了：前者负责将本地采集得到的音视频数据使用Socket传输给远程端；而后者使用Socket接收远程端发送过来的音视频数据，然后在本地播放。

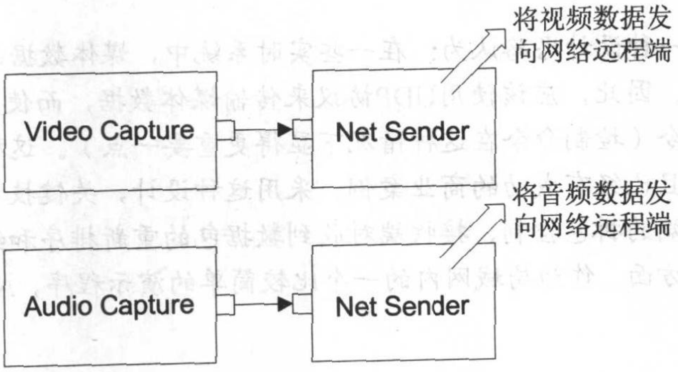


图 3.15 将本地采集的音视频发向网络远程端

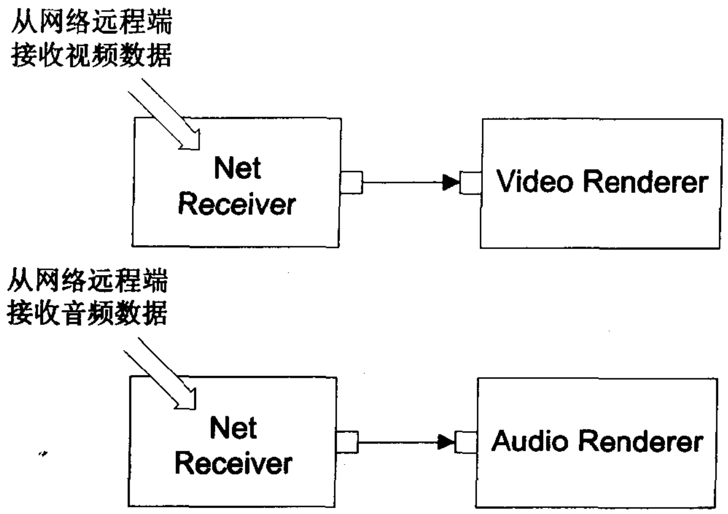


图 3.16　播放从网络远程端接收的音视频

图3.15中的Net Sender和图3.16中的Net Receiver是两个必须自行开发的Filter。构建如图3.15和图3.16所示的Filter Graph并不难，但是在什么时候构建它们呢？Net Receiver怎么知道将要接收的数据格式（以便与Renderer Filter建立连接）呢？从远程端接收到的音视频数据，在本地播放时如何保持同步？如何减小延迟？使用哪种网络协议来传输音视频数据？通信双方如何相互控制？需要建立几条Socket连接用于数据的传输？Socket相关的操作放在Net Sender和Net Receiver内实现，还是放在应用程序层上实现？这些问题都是亟待解决的。

因为音视频数据是大量的、连续的、需要可靠传输的，因此这些数据的传输可以选用TCP协议。而通信双方的控制命令数据量很小，而且重要性也不是很强，因此可以选用UDP协议。为了让在Socket上传输的数据简单化或者说提高Socket传输的效率，AVChat程序将音频数据和视频数据的传输分离，分别为它们创建一条Socket连接。而在单个Socket上，可以同时进行数据的发送和接收。

提示： 有一种设计思路认为：在一些实时系统中，媒体数据的实时性比正确性更重要。因此，应该使用UDP协议来传输媒体数据，而使用TCP协议来传输控制命令（控制命令在这种情况下显得更重要一点）。这种设计思路是可取的，而且已经有成功的商业案例。采用这种设计，关键技术还在于媒体数据在发送端的拆包控制、接收端对收到数据包的重新排序和组包、数据包丢失处理等方面。作为局域网内的一个比较简单的演示程序，AVChat不采用这种设计。

作为Filter开发的一条基本原则，Filter实现的功能应该力求单一。因此AVChat程序将Socket相关的操作都放在应用程序层上完成，而仅将完成连接的Socket句柄通过Filter的公共接口设置给Net Sender和Net Receiver。这两个Filter在内部实现时，只须使用外部设置进来的Socket进行数据的发送和接收就可以了，而不用去管Socket的创建、连接等一系列繁琐的交互过程。

当通信双方的Socket连接建立起来以后，构建图3.15的Filter Graph是没有任何问题的。

而此时构建图3.16的Filter Graph却不能完全成功（可以将Filter加入到Filter Graph中但不能完成连接），因为Net Receiver不知道将要接收到何种格式的数据。可是Net Sender知道！（通过这个Filter与Capture Filter连接用的媒体类型可以知道）于是，当构建完成如图3.15所示的Filter Graph后，应该马上将它运行起来。然后在Net Sender接收到Capture Filter发送出来的第一个Sample时，首先将数据格式发送给远程端；远程端在接收到这个数据格式之后，就可以重构一个DirectShow中使用的媒体类型，然后通知应用程序完成Filter之间的连接。此时，运行图3.16的Filter Graph就能播放远程端的音视频了。

为了避免发送和接收、音频和视频的相互干扰，AVChat程序将音频的发送、视频的发送、音频的接收、视频的接收分离，分别为它们构建一个Filter Graph。也就是说，AVChat程序最多的情况下可能要同时构建4个Filter Graph。另外，在播放远程端的音视频时，可以不用考虑同步性问题，甚至可以不给Sample打时间戳，因为音视频的采集本来就是实时的、间隔均匀的。远程端的任务，只是在接收到本地采集发送过去的数据后，以最快的速度播放。

总之，AVChat程序（服务器和客户机）的交互流程设计如图3.17所示。

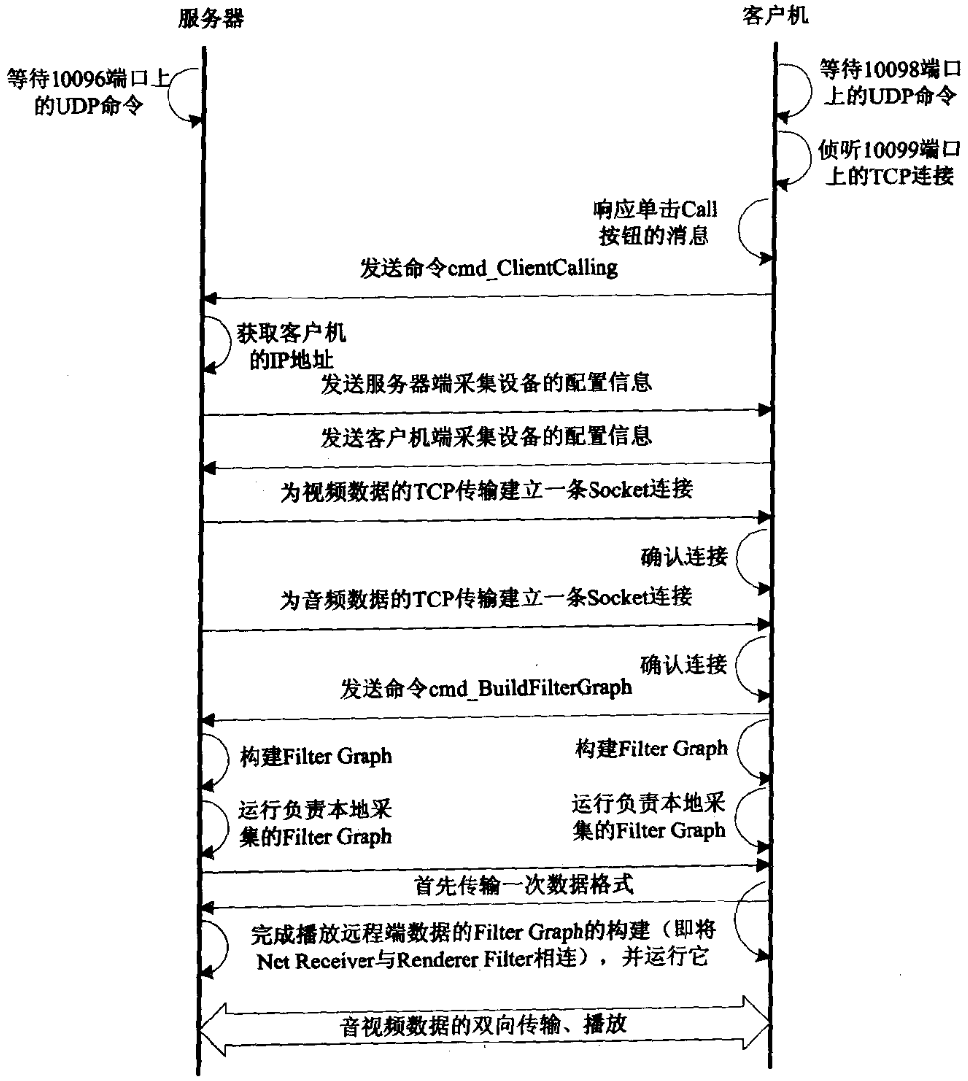


图 3.17　AVChat 程序的交互流程

3.2.3 采集设备自检

在进入真正意义上的视频聊天阶段之前，通信双方最好能对本地的采集设备进行一下检测，并且把各自的配置情况告诉对方。因为如果本地没有视频采集设备，那么远程端的用于接收网络视频数据的Filter Graph就不用构建；音频的情况类似。反之亦然。

可以构建如图3.18的Filter Graph用于采集设备的自检。如果图中的音频和视频两条支路都能连接成功，说明本地的音视频采集设备能够正常工作。另外，构建如图3.18的Filter Graph还能用于本地采集参数的调整，如改变视频图像的显示参数（亮度、色调、饱和度等）、确认音频的输入端子（一般应该选择麦克风输入）等。通过视频图像的预览，还可以调整摄像头的方位以对准目标，保证传送给远程端的图像是你想要传送的。

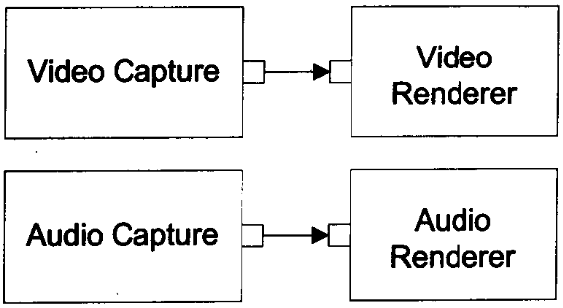


图 3.18　用于自检的 Filter Graph

既然要创建Filter Graph，那么最好先写一个控制类。这个控制类的实现要兼顾到以后用于音视频传送/接收用的Filter Graph的构建，以减少重复编码。具体实现中，AVChat程序使用了一个CPreviewAdmin类。这个类的定义和实现如下：

```
//
// CPreviewAdmin.h
//

#ifndef __H_CPreviewAdmin__
#define __H_CPreviewAdmin__

class CDXGraph;
class CDXFilter;
class CAVDevice;
class CVideoWindow;
class CPreviewAdmin
{
protected:
    CDXGraph *      mGraph;                 // Filter Graph封装类
    CVideoWindow *      mVideoWindow;       // 视频显示窗口
    CAVDevice *     mVideoDevice;           // 视频采集设备信息
    CAVDevice *     mAudioDevice;           // 音频采集设备信息
    CDXFilter *     mVideoFilter;           // 视频采集设备Filter
    CDXFilter *     mAudioFilter;           // 音频采集设备Filter
    CDXFilter *     mVideoRenderer;         // Video Renderer
    CDXFilter *     mAudioRenderer;         // Audio Renderer
```

```
public:
    CPreviewAdmin();
    virtual ~CPreviewAdmin();
    // 设置保存音视频采集设备信息的对象指针
    void SetAVDevice(CAVDevice * inVideo, CAVDevice * inAudio);
    // 设置视频显示窗口
    void SetVideoWindow(CVideoWindow * inWindow);
    // 构建/销毁Filter Graph
    virtual BOOL Activate(void);
    virtual void Deactivate(void);
    // Filter Graph的控制：运行、暂停、停止
    BOOL PlayGraph(void);
    BOOL PauseGraph(void);
    virtual BOOL StopGraph(void);
    // 得到音视频采集设备Filter
    CDXFilter * GetVideoInputFilter(void) {return mVideoFilter;}
    CDXFilter * GetAudioInputFilter(void) {return mAudioFilter;}
    // 显示音视频采集设备Filter的属性页
    void ShowVideoCapturePropertyPage(void);
    void ShowAudioCapturePropertyPage(void);
protected:
    virtual BOOL CreateGraph(void); // 创建Filter Graph
    virtual void DeleteGraph(void); // 销毁Filter Graph
    // 输入部分Filter的创建和销毁
    virtual BOOL CreateInputFilters(void);
    virtual void ReleaseInputFilters(void);
    // 输出部分Filter的创建和销毁
    virtual BOOL CreateOutputFilters(void);
    virtual void ReleaseOutputFilters(void);
    // 将输入部分的Filter和输出部分的Filter相连
    virtual BOOL ConnectFilters(void);
    // 当Filter Graph创建、连接成功后进行一些必要的参数设置
    virtual BOOL AfterBuildGraph(void);
    // 工具函数：将数码摄像机的视频输出Pin连接到指定的输入Pin上，中间会插入一个DV
        Video Decoder Filter
    BOOL RenderDVVideo(CDXGraph * inGraph, IPin * inDVPin, IPin *
        inRenderedPin);
};

// 几个很实用的宏定义
#define SAFE_DELETE(p)       { if(p) { delete (p);     (p)=NULL; } }
#define SAFE_DELETE_ARRAY(p) { if(p) { delete[] (p);   (p)=NULL; } }
#define SAFE_RELEASE(p)      { if(p) { (p)->Release(); (p)=NULL; } }

#endif // __H_CPreviewAdmin__

//
// CPreviewAdmin.cpp
//

#include "stdafx.h"
#include <streams.h>
```

```
#include "CPreviewAdmin.h"
#include "CDXGraph.h"
#include "CDXFilter.h"
#include "CAVDevice.h"
#include "CVideoWindow.h"
#include "GlobalDefs.h"
#include "UDsUtils.h"
#include "CVideoCaptureFilter2.h"
#include "CAudioCaptureFilter2.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//////////////////////////////////////////////////////////////////////
CPreviewAdmin::CPreviewAdmin()
{
    // 初始化
    mGraph          = NULL;
    mVideoWindow    = NULL;
    mVideoDevice    = NULL;
    mAudioDevice    = NULL;
    mVideoFilter    = NULL;
    mAudioFilter    = NULL;
    mVideoRenderer  = NULL;
    mAudioRenderer  = NULL;
}
// 在析构函数中记得销毁Filter Graph
CPreviewAdmin::~CPreviewAdmin()
{
    Deactivate();
}
// 设置保存音视频采集设备信息的对象指针
void CPreviewAdmin::SetAVDevice(CAVDevice * inVideo, CAVDevice * inAudio)
{
    mVideoDevice = inVideo;
    mAudioDevice = inAudio;
}
// 设置视频显示窗口
void CPreviewAdmin::SetVideoWindow(CVideoWindow * inWindow)
{
    mVideoWindow = inWindow;
}
// 构建Filter Graph，包括创建Filter Graph、输入部分的Filter、输出部分的Filter，
// 并将输入部分的Filter与输出部分的Filter相连，以构成完整的链路
BOOL CPreviewAdmin::Activate(void)
{
    BOOL pass = CreateGraph();  // 创建一个Filter Graph
    if (pass)
    {
```

```
        pass = CreateInputFilters();  // 创建输入部分的Filter
    }
    if (pass)
    {
        pass = CreateOutputFilters(); // 创建输出部分的Filter
    }
    if (pass)
    {
        // 将输入部分的Filter与输出部分的Filter相连
        pass = ConnectFilters();
    }
    if (pass)
    {
        // 进行一些“善后”处理
        pass = AfterBuildGraph();
    }
    return pass;
}
// 销毁Filter Graph
void CPreviewAdmin::Deactivate(void)
{
    // 首先停止Filter Graph的运行
    StopGraph();
    // 销毁输入部分、输出部分的所有Filter
    ReleaseInputFilters();
    ReleaseOutputFilters();
    //最后销毁Filter Graph
    DeleteGraph();
}
// 创建Filter Graph
BOOL CPreviewAdmin::CreateGraph(void)
{
    mGraph = new CDXGraph();
    BOOL pass = mGraph->Create();
    return pass;
}
// 销毁Filter Graph
void CPreviewAdmin::DeleteGraph(void)
{
    if (mVideoWindow)
    {
        mVideoWindow->SetGraph(0);
    }
    SAFE_DELETE(mGraph);
}
// 创建输入部分的Filter（音视频采集设备Filter）
BOOL CPreviewAdmin::CreateInputFilters(void)
{
    BOOL pass = FALSE;
    // 创建视频采集设备Filter，并且获取成功创建的设备信息
    mVideoFilter = new CVideoCaptureFilter2(mGraph->GetGraph());
    ((CVideoCaptureFilter2 *)mVideoFilter)->SetDevice(mVideoDevice);
```

```
        pass |= mVideoFilter->CreateFilter();
        // 创建音频采集设备Filter，并且获取成功创建的设备信息
        mAudioFilter = new CAudioCaptureFilter2(mGraph->GetGraph());
        ((CAudioCaptureFilter2 *)mAudioFilter)->SetDevice(mAudioDevice);
        pass |= mAudioFilter->CreateFilter();
        return pass;
}
// 销毁输入部分的Filter (音视频采集设备Filter)
void CPreviewAdmin::ReleaseInputFilters(void)
{
        SAFE_DELETE(mVideoFilter);
        SAFE_DELETE(mAudioFilter);
}
// 创建输出部分的Filter (Video Renderer和Audio Renderer)
BOOL CPreviewAdmin::CreateOutputFilters(void)
{
        BOOL pass = TRUE;
        // 创建Video Renderer
        mVideoRenderer = new CDXFilter(mGraph->GetGraph(),
            CLSID_VideoRenderer, "Video Renderer");
        pass = mVideoRenderer->CreateFilter();
        // 创建Audio Renderer
        mAudioRenderer = new CDXFilter(mGraph->GetGraph(),
            CLSID_DSoundRender, "Audio Renderer");
        pass = mAudioRenderer->CreateFilter();

        return pass;
}
// 销毁输出部分的Filter (Video Renderer和Audio Renderer)
void CPreviewAdmin::ReleaseOutputFilters(void)
{
        SAFE_DELETE(mVideoRenderer);
        SAFE_DELETE(mAudioRenderer);
}
// 将输入部分的Filter与输出部分的Filter相连
BOOL CPreviewAdmin::ConnectFilters(void)
{
        BOOL pass   = FALSE;
        IPin * pOut = NULL;
        IPin * pIn  = NULL;
        // 连接音频采集设备Filter到Audio Renderer
        if (mAudioDevice->IsDeviceAvailable())
        {
            pOut = mAudioFilter->GetPin(FALSE);
            pIn  = mAudioRenderer->GetPin(TRUE);
            pass = mGraph->ConnectFilters(pOut, pIn);
        }
        // 连接视频采集设备Filter到Video Renderer
        if (pass && mVideoDevice->IsDeviceAvailable())
        {
            if (mVideoDevice->IsDVDevice())  // 视频采集设备是数码摄像机
            {
```

```
            pOut = mVideoFilter->GetPin(FALSE, "DV Vid Out");
            pIn  = mVideoRenderer->GetPin(TRUE);
            // 使用自定义函数，将DV Vid Out输出Pin连接到Video Renderer上
            pass = RenderDVVideo(mGraph, pOut, pIn);
        }
        else   // 视频采集设备是USB摄像头
        {
            pOut = mVideoFilter->GetPin(FALSE, "Capture");
            pIn  = mVideoRenderer->GetPin(TRUE);
            // 使用智能连接，将Capture输出Pin连接到Video Renderer上
            HRESULT hr = mGraph->GetGraph()->Connect(pOut, pIn);
            pass = SUCCEEDED(hr);
        }
    }
    return pass;
}
// 当Filter Graph完成构建后，设置视频显示窗口，并自动运行Filter Graph
BOOL CPreviewAdmin::AfterBuildGraph(void)
{
    if (mVideoWindow)
    {
        mVideoWindow->SetGraph(mGraph);
        // 自动运行Filter Graph
        PlayGraph();
        return TRUE;
    }
    return FALSE;
}
// 运行Filter Graph
BOOL CPreviewAdmin::PlayGraph(void)
{
    if (mGraph)
    {
        mGraph->Run();
        return TRUE;
    }
    return FALSE;
}
// 暂停Filter Graph
BOOL CPreviewAdmin::PauseGraph(void)
{
    if (mGraph)
    {
        mGraph->Pause();
        return TRUE;
    }
    return FALSE;
}
// 停止Filter Graph
BOOL CPreviewAdmin::StopGraph(void)
{
    if (mGraph)
```

```
	{
		mGraph->Stop();
		return TRUE;
	}
	return FALSE;
}
// 显示视频采集设备Filter的属性页
void CPreviewAdmin::ShowVideoCapturePropertyPage(void)
{
	// 如果视频采集设备是数码摄像机，则不显示属性页
	if (!mVideoDevice->IsDVDevice())
	{
		UDsUtils::ShowFilterPropertyPage(mVideoFilter->GetFilter());
		// 当采集设备属性页窗口关闭后，更新视频的解码制式
		long format = ((CVideoCaptureFilter2
			*)mVideoFilter)->GetResolution();
		mVideoDevice->SetVideoResolution(format);
	}
}
// 显示音频采集设备Filter的属性页
void CPreviewAdmin::ShowAudioCapturePropertyPage(void)
{
	UDsUtils::ShowFilterPropertyPage(mAudioFilter->GetFilter());
	// 当采集设备属性页窗口关闭后，更新音频的输入端子
	long index = ((CAudioCaptureFilter2 *)mAudioFilter)->GetConnector();
	mAudioDevice->SetAudioConnector(index);
}
// 工具函数：将数码摄像机的视频输出Pin连接到指定的输入Pin上
BOOL CPreviewAdmin::RenderDVVideo(CDXGraph * inGraph, IPin * inDVPin, IPin
	* inRenderedPin)
{
	BOOL pass   = FALSE;
	IPin * pOut = NULL;
	IPin * pIn  = NULL;
	if (inGraph)
	{
		// 创建DV Video Decoder Filter
		CDXFilter * pDecoder = NULL;
		pDecoder = new CDXFilter(inGraph->GetGraph(), CLSID_DVVideoCodec,
			"DV Video Decoder");
		pDecoder->CreateFilter();
		// 在DV Video Decoder上设置输出视频图像的尺寸
		IIPDVDec *  pIPDVDec = NULL;
		pDecoder->QueryInterface(IID_IIPDVDec, (void**)&pIPDVDec);
		if (pIPDVDec)
		{
			pIPDVDec->Release();
			// DVRESOLUTION_HALF? DVRESOLUTION_QUARTER?
			pIPDVDec->put_IPDisplay(DVRESOLUTION_HALF);
		}
		// 将数码摄像机的视频输出Pin连接到DV Video Decoder
		pIn  = pDecoder->GetPin(TRUE);
```

```
        pass = inGraph->ConnectFilters(inDVPin, pIn);
        // 将DV Video Decoder连接到指定的输入Pin上
        pOut = pDecoder->GetPin(FALSE);
        pass = inGraph->ConnectFilters(pOut, inRenderedPin);
        // 销毁DV Video Decoder Filter的包装对象
        pDecoder->Detach();
        delete pDecoder;
    }
    return pass;
}
```

可以看出，CPreviewAdmin类在实现时用到了一系列的包装类。这些包装类大部分都已经在本书的第2章中介绍过了，读者现在对它们应该很熟悉了。即使有些类是第一次出现，它们也不是全新的，读者总可以到第2章的AVCap程序中找到它们的影子，如CAVDevice对应于CDSDevice、CAudioCaptureFilter2对应于CAudioCaptureFilter、CVideoCaptureFilter2对应于CVideoCaptureFilter等等。事实上，AVChat程序用到的很多类都是从AVCap程序中借鉴过来的，毕竟这两个程序在音视频采集部分的功能实现上是类似的。有些类照搬过来了，有些则做了细小的调整，读者有兴趣的话可以进行一下对比。

大家已经知道，音视频采集设备Filter的创建必须通过系统枚举来完成。这部分的实现已经在本书的2.2.2节中作出了详细的介绍。这里还有一点需要强调的是，AVChat程序并不想创建那些专门用于音视频采集的采集卡（而只想使用数码摄像机或USB摄像头），那么如何来做出这种限制呢？

```
//
// CVideoCaptureFilter2.cpp
//
BOOL CVideoCaptureFilter2::CreateFilter(void)
{
    if (mFilter)
    {
        return TRUE;
    }
    ASSERT(mDevice);
    if (mGraph)
    {
        mFilter = UDsUtils::CreateCamera(*mDevice);
        // 其他代码省略
        // ……
}
// ……
}

//
// UDsUtils.cpp
//
// 创建视频采集设备Filter（并把成功创建的设备信息保存到outDevice对象中）
IBaseFilter * UDsUtils::CreateCamera(CAVDevice& outDevice)
{
    return CreateAVDevice(CLSID_VideoInputDeviceCategory, outDevice);
```

```
}
// 创建音视频采集设备（并把成功创建的设备信息保存到outDevice对象中）
IBaseFilter * UDsUtils::CreateAVDevice(GUID inCategory, CAVDevice&
    outDevice)
{
    IBaseFilter * pDevice = NULL;
    // 系统设备枚举（代码省略）
    // ……

    IBaseFilter * filter = NULL;
    // 将枚举得到的设备标识绑定成Filter的形式
    if (SUCCEEDED(moniker->BindToObject(0, 0, IID_IBaseFilter,
        (void**)&filter)))
    {
        // 注意:判断设备的类型
        Device_Type type = DetermineDeviceType(filter);
        if (type != DT_Unknown)
        {
            // 如果不是未知设备，则保存设备信息
            outDevice.SetDeviceDisplayName(displayName);
            outDevice.SetDeviceFriendlyName(friendlyName);
            outDevice.SetDevideType(type);
            // 返回这个Filter作为成功创建的采集设备
            pDevice = filter;
            pDevice->AddRef();
        }
        filter->Release();
    }

    // ……
    return pDevice;
}
// 判断采集设备的类型：仅让数码摄像机、USB摄像头、音频采集设备通过
Device_Type UDsUtils::DetermineDeviceType(IBaseFilter * inFilter)
{
    // 如果Filter上支持IAMExtTransport接口，则认为该设备是数码摄像机
    IAMExtTransport * pAMExtTransPost = NULL;
    inFilter->QueryInterface(IID_IAMExtTransport, (
        void **)&pAMExtTransPost);
    if (pAMExtTransPost)
    {
        pAMExtTransPost->Release();
        return DT_DV;
    }
    // 如果Filter上支持IAMAnalogVideoDecoder或IAMCameraControl或IAMVideo-
    // ProcAmp接口，则认为该设备是视频采集设备
    IAMAnalogVideoDecoder * pDecoder     = NULL;
    IAMCameraControl *      pCamera      = NULL;
    IAMVideoProcAmp *       pAmp         = NULL;
    inFilter->QueryInterface(IID_IAMAnalogVideoDecoder,
        (void **)&pDecoder);
    inFilter->QueryInterface(IID_IAMCameraControl, (void **)&pCamera);
```

```
        inFilter->QueryInterface(IID_IAMVideoProcAmp, (void **)&pAmp);
        if (pDecoder || pCamera || pAmp)
        {
            SAFE_RELEASE(pDecoder);
            SAFE_RELEASE(pCamera);
            SAFE_RELEASE(pAmp);
            // 进一步判断是否是USB摄像头
            // 因为代表USB摄像头的Capture Filter通常比较简单，所以Filter上Pin的数量
            // 不会超过2个
            int inputs = 0, outputs = 0;
            UFilterUtils::GetPinCount(inFilter, inputs, outputs);
            if (inputs + outputs <= 2)
            {
                return DT_USB_Camera;
            }
            else
            {
              // 如果Filter上Pin的数量超过2个，则标记这个设备为一个未知设备，不支持使用
              return DT_Unknown;
            }
        }
        // 如果Filter上支持IAMAudioInputMixer接口，则认为该设备是音频采集设备
        IAMAudioInputMixer * pAudioMixer = NULL;
        inFilter->QueryInterface(IID_IAMAudioInputMixer, (void **) &pAudio
            Mixer);
        if (pAudioMixer)
        {
            pAudioMixer->Release();
            return DT_Audio_Capture;
        }
        // 如果上述判断均不成立，则认为这是一个未知设备，将得不到支持
        return DT_Unknown;
    }
```

接下来就是CPreviewAdmin类在AVChat程序中的使用问题了。采集设备的自检使用一个对话框界面来操作，这个对话框类为CSelfCheckDlg。在这个界面上，不仅可以对本地采集设备进行检测以及参数调整，用户还可以根据自己的意愿来设置是否将本地的音频或者视频发送给远程端（这样可以使得AVChat程序提供的服务更加灵活）。下面列出了CSelfCheckDlg类的定义和实现：

```
#if !defined(AFX_CSELFCHECKDLG_H__B3B2CC54_7B5B_4F6D_94CB_CD88B9DA03F0__
    INCLUDED_)
#define AFX_CSELFCHECKDLG_H__B3B2CC54_7B5B_4F6D_94CB_CD88B9DA03F0__
    INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// CSelfCheckDlg.h : header file
//
```

```
#include "CPreviewAdmin.h"
#include "CVideoWindow.h"

class CSelfCheckDlg : public CDialog
{
// Construction
public:
    CSelfCheckDlg(CWnd* pParent = NULL);   // standard constructor
// Dialog Data
    //{{AFX_DATA(CSelfCheckDlg)
    enum { IDD = IDD_ADJUST_DIALOG };
    CComboBox   mComboAudioInputs;      // 音频输入端子组合框
    CVideoWindow    mVideoWindow;       // 视频显示窗口
    //}}AFX_DATA
// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CSelfCheckDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL
public:
    // 设置保存音视频采集设备信息的对象指针
    void SetAVDevice(CAVDevice * inVideo, CAVDevice * inAudio);
    // 获得本地采集设备的配置情况
    long GetDeviceConfig(void);
// Implementation
protected:
    CAVDevice *     mVideoDevice;       // 视频采集设备信息
    CAVDevice *     mAudioDevice;       // 音频采集设备信息
    long            mDeviceConfig;      // 设备配置情况
    CPreviewAdmin       mDeviceAdmin;   // Filter Graph控制类对象
    // 将音频输入端子填入界面上的一个组合框内
    void FillAudioConnectors(void);

    // Generated message map functions
    //{{AFX_MSG(CSelfCheckDlg)
    virtual BOOL OnInitDialog();
    afx_msg void OnButtonAudioCapture();
    afx_msg void OnButtonVideoCapture();
    afx_msg void OnSelchangeComboAudioInputs();
    afx_msg void OnDestroy();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
#endif
// !defined(AFX_CSELFCHECKDLG_H__B3B2CC54_7B5B_4F6D_94CB_
//CD88B9DA03F0__INCLUDED_)

// CSelfCheckDlg.cpp : implementation file
//
```

```
#include "stdafx.h"
#include <streams.h>
#include "AVChat.h"
#include "CSelfCheckDlg.h"
#include "CAVDevice.h"
#include "CAudioCaptureFilter2.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
CSelfCheckDlg::CSelfCheckDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CSelfCheckDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CSelfCheckDlg)
        // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
    mVideoDevice = NULL;
    mAudioDevice = NULL;
    mDeviceConfig = 0;
}
void CSelfCheckDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CSelfCheckDlg)
    DDX_Control(pDX, IDC_COMBO_AUDIO_INPUTS, mComboAudioInputs);
    DDX_Control(pDX, IDC_ADJUST_WINDOW, mVideoWindow);
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CSelfCheckDlg, CDialog)
    //{{AFX_MSG_MAP(CSelfCheckDlg)
    ON_BN_CLICKED(IDC_BUTTON_AUDIO_CAPTURE, OnButtonAudioCapture)
    ON_BN_CLICKED(IDC_BUTTON_VIDEO_CAPTURE, OnButtonVideoCapture)
    ON_CBN_SELCHANGE(IDC_COMBO_AUDIO_INPUTS,
        OnSelchangeComboAudioInputs)
    ON_WM_DESTROY()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// 设置保存音视频采集设备信息的对象指针
void CSelfCheckDlg::SetAVDevice(CAVDevice * inVideo, CAVDevice * inAudio)
{
    mVideoDevice = inVideo;
    mAudioDevice = inAudio;
    mDeviceAdmin.SetAVDevice(inVideo, inAudio);
}
```

```
// 对话框初始化函数
BOOL CSelfCheckDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // 设置视频显示窗口
    mDeviceAdmin.SetVideoWindow(&mVideoWindow);
    // 构建Filter Graph
    if (mDeviceAdmin.Activate())
    {
        // 获得所有音频输入端子，并填入界面上的一个组合框
        FillAudioConnectors();
    }

    return TRUE;    // return TRUE unless you set the focus to a control
                    // EXCEPTION: OCX Property Pages should return FALSE
}
// 将音频输入端子填入界面上的一个组合框内，供用户选择
void CSelfCheckDlg::FillAudioConnectors(void)
{
    mComboAudioInputs.ResetContent();
    // 获得音频采集设备Filter
    CAudioCaptureFilter2 * pFilter = NULL;
    pFilter = (CAudioCaptureFilter2 *) mDeviceAdmin.GetAudioInputFilter();
    if (pFilter)
    {
        // 获取音频输入端子的个数
        long count = pFilter->GetConnectorCount();
        for (long i = 0; i < count; i++)
        {
            // 将各个音频输入端子的名字加入组合框中
            mComboAudioInputs.AddString(pFilter->GetConnectorName(i));
        }
        mComboAudioInputs.SetCurSel(pFilter->GetConnector());
    }
}
// 用户选择了一个音频输入端子的消息响应
void CSelfCheckDlg::OnSelchangeComboAudioInputs()
{
    int index = mComboAudioInputs.GetCurSel();
    if (index >= 0)
    {
        // 将用户的选择设置给音频采集设备Filter
        CAudioCaptureFilter2 * pFilter = NULL;
        pFilter = (CAudioCaptureFilter2 *) mDeviceAdmin.GetAudioInput
            Filter();
        if (pFilter)
        {
            pFilter->SetConnector(index);
            // 保存用户的选择
            mAudioDevice->SetAudioConnector(index);
        }
```

```
    }
}
// 显示音频采集设备Filter的属性页
void CSelfCheckDlg::OnButtonAudioCapture()
{
    mDeviceAdmin.ShowAudioCapturePropertyPage();
}
// 显示视频采集设备Filter的属性页
void CSelfCheckDlg::OnButtonVideoCapture()
{
    mDeviceAdmin.ShowVideoCapturePropertyPage();
}
// 获得本地采集设备的配置情况
long CSelfCheckDlg::GetDeviceConfig(void)
{
    return mDeviceConfig;
}
// 当这个自检对话框销毁时，进行采集设备配置情况的更新
void CSelfCheckDlg::OnDestroy()
{
    // 销毁Filter Graph
    mDeviceAdmin.Deactivate();
    // 如果本地视频采集设备能够正常工作，而且用户允许将视频传送给远程端，将设置一个本
    // 地拥有视频采集设备的标记
    int isVideoDisable = IsDlgButtonChecked(IDC_CHECK_VIDEO_DISABLE);
    if (mVideoDevice->IsDeviceAvailable() && !isVideoDisable)
    {
        mDeviceConfig |= Local_Has_Video;
    }
    // 如果本地音频采集设备能够正常工作，而且用户允许将音频传送给远程端，将设置一个本
    // 地拥有音频采集设备的标记
    int isAudioDisable = IsDlgButtonChecked(IDC_CHECK_AUDIO_DISABLE);
    if (mAudioDevice->IsDeviceAvailable() && !isAudioDisable)
    {
        mDeviceConfig |= Local_Has_Audio;
    }

    CDialog::OnDestroy();
}
```

自检对话框在主界面中的使用如下：

```
// AVChatDlg.cpp : implementation file
//
// 初始化函数
void CAVChatDlg::Initialize(void)
{
    // 通过弹出角色选择对话框让用户选择一个角色，然后根据用户的选择创建一个角色控制类
    // 对象mActor
    // 代码省略……

    // 弹出一个自检对话框让用户进行音视频采集设备的调整
    CSelfCheckDlg chkDlg;
```

```
    chkDlg.SetAVDevice(&mVideoDevice, &mAudioDevice);
    chkDlg.DoModal();
    // 获取本地采集设备的配置情况，然后设置给角色控制类对象mActor
    mActor->SetDeviceConfig(chkDlg.GetDeviceConfig());
    // 更新界面
    ReflectStatus();
}
```

3.2.4 角色控制实现

在建立网络连接之前，通信双方扮演的角色是不一样的，而且必须一个是服务器，一个是客户机。于是出现了问题：角色不同，要求的控制逻辑也不会一样，但要在同一个程序中实现。这显然对AVChat程序的设计提出了更高的要求。

幸运的是，AVChat程序做到了！它使用一个CRoleAdmin类来实现服务器和客户机两种角色控制的共性，然后以CRoleAdmin为父类，派生一个CServerAdmin类专门定制服务器的特性，派生一个CClientAdmin类专门定制客户机的特性。考虑到角色控制同样也包括对Filter Graph的控制，因此将CRoleAdmin类从CPreviewAdmin类派生。

关于服务器和客户机之间的通信，AVChat程序将监听TCP连接的Socket放在客户机程序，传输控制命令使用UDP协议，而传输媒体数据使用TCP协议。传输的UDP数据和TCP数据都有自己定义的格式。UDP数据包定义为如下格式：

```
//
// GlobalDefs.h
//
struct UDP_Pack
{
    long command;    // 命令类型
    long param1;     // 命令携带的第1个参数
    long param2;     // 命令携带的第2个参数
    long reserved;   // 保留

    // 将上述各变量值从主机字节顺序转换成网络字节顺序
    void my_hton(void)
    {
        command     = htonl(command);
        param1      = htonl(param1);
        param2      = htonl(param2);
        reserved    = htonl(reserved);
    };
    // 将上述各变量值从网络字节顺序转换成主机字节顺序
    void my_ntoh(void)
    {
        command     = ntohl(command);
        param1      = ntohl(param1);
        param2      = ntohl(param2);
        reserved    = ntohl(reserved);
    };
};
// 各种命令类型定义
```

```
const long MAX_COMMAND_SIZE         = 100;  // UDP命令的最大大小
const long cmd_ClientCalling        = 'avct' + 100; // 客户机要求连接
const long cmd_DeviceConfig         = 'avct' + 101; // 设备配置信息
const long cmd_BuildFilterGraph     = 'avct' + 102; // 要求远程端创建Filter
                                                    // Graph
const long cmd_DisconnectRequest    = 'avct' + 103; // 要求远程端断开连接
```

TCP数据包的包头定义为如下格式（固定大小的包头后面跟负载数据）：

```
struct Pack_Header
{
    long pack_type;  // 数据包的类型：格式数据还是媒体数据
    long pack_size;  // 数据包的大小（不包括本包头的大小）

    // 将上述各变量值从主机字节顺序转换成网络字节顺序
    void my_hton(void)
    {
        pack_type       = htonl(pack_type);
        pack_size       = htonl(pack_size);
    };
    // 将上述各变量值从网络字节顺序转换成主机字节顺序
    void my_ntoh(void)
    {
        pack_type       = ntohl(pack_type);
        pack_size       = ntohl(pack_size);
    };
};
// TCP数据包类型定义
const long PT_AudioMediaType     = 10001;    // 音频格式数据
const long PT_VideoMediaType     = 10002;    // 视频格式数据
const long PT_Payload            = 10003;    // 音频或视频媒体数据
```

下面，就来重点介绍角色控制类CRoleAdmin、CServerAdmin、CClientAdmin的实现。

提示：CRoleAdmin类不仅从CPreviewAdmin类派生，它同时还从CMsgStation类和CMsgReceiver类派生。关于CMsgStation和CMsgReceiver两个类的功能介绍，请读者参见2.4.1节。

```
//
// CRoleAdmin.h
//

#ifndef __H_CRoleAdmin__
#define __H_CRoleAdmin__

#include "CPreviewAdmin.h"
#include "CUDPManager.h"
#include "CMsgReceiver.h"
#include "CMsgStation.h"

class CDXGraph;
class CFilterNetReceiver;
```

```
class CFilterNetSender;
class CRoleAdmin : public CPreviewAdmin, public CMsgReceiver, public
    CMsgStation
{
protected:
    long            mDeviceConfig;  // 设备的配置情况（包括本地的和远程的）
    CUDPManager     mUdpManager;    // UDP控制对象，用于发送、接收命令
    WORD            mLocalUDPPort;  // 本地UDP端口
    WORD            mTargetUDPPort; // 远程UDP端口
    DWORD           mTargetIP;      // 远程IP地址

    CDXGraph *      mRemoteVideoGraph;  // 播放远程视频的Filter Graph
    CDXGraph *      mRemoteAudioGraph;  // 播放远程音频的Filter Graph
    CDXGraph *      mLocalVideoGraph;   // 发送本地视频的Filter Graph
    CDXGraph *      mLocalAudioGraph;   // 发送本地音频的Filter Graph

    CFilterNetReceiver *    mVideoReceiver;     // 接收远程视频数据的Filter
    CFilterNetReceiver *    mAudioReceiver;     // 接收远程音频数据的Filter
    CFilterNetSender *      mVideoSender;       // 发送本地视频数据的Filter
    CFilterNetSender *      mAudioSender;       // 发送本地音频数据的Filter
public:
    CRoleAdmin();
    virtual ~CRoleAdmin();
    // 设置采集设备配置情况
    void SetDeviceConfig(long inConfig);
    // 设置远程端的IP地址
    void SetTargetIP(DWORD inTarget);
    // 初始化/反初始化
    virtual BOOL Init(void);
    virtual void Uninit(void);
    // 客户机呼叫服务器
    virtual BOOL CallServer(void);
    // 断开通话连接
    virtual void Disconnect(BOOL inNotifyPeer = TRUE);
    // 停止Filter Graph的运行（CPreviewAdmin类的函数）
    virtual BOOL StopGraph(void);
protected:
    // 处理自定义消息
    virtual bool ReceiveMessage(MessageT inMessage, void * ioParam, void *
        ioParam2);
    // 发送简单命令（不带参数的命令）
    BOOL SendSimpleCommand(long inCommand);
    // 将本地的采集设备配置信息发送给远程端
    BOOL SendLocalDeviceConfigToPeer(void);
    // 完成播放远程视频的Filter Graph的构建
    void CompleteVideoReceivingGraph(void);
    // 完成播放远程音频的Filter Graph的构建
    void CompleteAudioReceivingGraph(void);
    // 创建/销毁Filter Graph
    virtual BOOL CreateGraph(void);
    virtual void DeleteGraph(void);
    // 创建/销毁输入部分的Filter
```

```
    virtual BOOL CreateInputFilters(void);
    virtual void ReleaseInputFilters(void);
    // 创建/销毁输出部分的Filter
    virtual BOOL CreateOutputFilters(void);
    virtual void ReleaseOutputFilters(void);
    // 连接输入部分和输出部分的Filter
    virtual BOOL ConnectFilters(void);
    // 当完成Filter Graph的构建之后进行一些操作
    virtual BOOL AfterBuildGraph(void);
    // 得到用于视频传输的Socket
    virtual SOCKET GetVideoStreamSocket(void);
    // 得到用于音频传输的Socket
    virtual SOCKET GetAudioStreamSocket(void);
};

#endif // __H_CRoleAdmin__

//
// CRoleAdmin.cpp
//

#include "stdafx.h"
#include <streams.h>
#include "CRoleAdmin.h"
#include "GlobalDefs.h"
#include "CDXGraph.h"
#include "CVideoWindow.h"
#include "CAVDevice.h"
#include "CVideoCaptureFilter2.h"
#include "CAudioCaptureFilter2.h"
#include "CFilterNetReceiver.h"
#include "CFilterNetSender.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

///////////////////////////////////////////////////////////////////////////
CRoleAdmin::CRoleAdmin() : mDeviceConfig(0), mLocalUDPPort(0),
mTargetUDPPort(0), mTargetIP(0x7f000001)
{
    // 参数初始化
    mRemoteVideoGraph   = NULL;
    mRemoteAudioGraph   = NULL;
    mLocalVideoGraph    = NULL;
    mLocalAudioGraph    = NULL;
    mVideoReceiver      = NULL;
    mAudioReceiver      = NULL;
    mVideoSender        = NULL;
    mAudioSender        = NULL;
```

```
    // 监听来自UDP命令控制对象的消息
    mUdpManager.AddMsgReceiver(this);
}
CRoleAdmin::~CRoleAdmin()
{
    // 释放资源
    Deactivate();
    Uninit();
}
// 设置采集设备配置情况（将本地端和远程端的采集设备信息合并到同一个变量中）
/*
typedef enum
{
    Local_Has_Video  = 1,  // 本地端有视频
    Local_Has_Audio  = 2,  // 本地端有音频
    Remote_Has_Video = 4,  // 远程端有视频
    Remote_Has_Audio = 8   // 远程端有音频
} Device_Config;
const long Local_Device_Mask    = 0x03; // 本地设备信息的屏蔽字
const long Remote_Device_Mask   = 0x0c; // 远程设备信息的屏蔽字
*/
void CRoleAdmin::SetDeviceConfig(long inConfig)
{
    mDeviceConfig |= inConfig;
}
// 设置远程端的IP地址
void CRoleAdmin::SetTargetIP(DWORD inTarget)
{
    mTargetIP = inTarget;
    mUdpManager.SetTargetIP(inTarget);
}
// 初始化函数
BOOL CRoleAdmin::Init(void)
{
    // 此时，远程端的IP地址可能还没有确定
    // 对于客户机，远程端（也就是服务器）的IP地址由用户指定
    // 对于服务器，远程端（也就是客户机）的IP地址由UDP命令控制对象自动检测
    // 设置本地端和远程端使用的UDP端口号
    mUdpManager.SetLocalPort(mLocalUDPPort);
    mUdpManager.SetTargetPort(mTargetUDPPort);
    // 创建UDP命令发送用的Socket
    mUdpManager.CreateSender();
    // 创建UDP命令接收用的Socket
    BOOL pass = mUdpManager.CreateReceiver();
    if (pass)
    {
        // 启动接收远程端UDP命令的子线程
        pass = mUdpManager.StartReceiving();
    }
    return pass;
}
// 反初始化函数
```

```
void CRoleAdmin::Uninit(void)
{
    // 停止接收远程端的UDP命令
    mUdpManager.StopReceiving();
}
// 呼叫服务器（供客户机子类定制）
BOOL CRoleAdmin::CallServer(void)
{
    return FALSE;
}
// 断开通话连接
void CRoleAdmin::Disconnect(BOOL inNotifyPeer)
{
    // 通知远程端将要断开连接,
    // 远程端在收到这个通知后可以销毁Filter Graph等资源
    if (inNotifyPeer)
    {
        SendSimpleCommand(cmd_DisconnectRequest);
    }
    // 销毁本地的Filter Graph等资源
    Deactivate();
    Uninit();
}
// 停止所有的Filter Graph
BOOL CRoleAdmin::StopGraph(void)
{
    if (mRemoteVideoGraph)
    {
        mRemoteVideoGraph->Stop();
    }
    if (mRemoteAudioGraph)
    {
        mRemoteAudioGraph->Stop();
    }
    if (mLocalVideoGraph)
    {
        mLocalVideoGraph->Stop();
    }
    if (mLocalAudioGraph)
    {
        mLocalAudioGraph->Stop();
    }
    return TRUE;
}
// 处理自定义消息
bool CRoleAdmin::ReceiveMessage(MessageT inMessage, void * ioParam, void *
    ioParam2)
{
    switch (inMessage)
    {
    case msg_MediaTypeReceived: // 接收到了一个媒体格式数据
        {
```

```
            // 完成播放远程音视频的Filter Graph的构建
            BOOL * isVideo = (BOOL*) ioParam;
            if (*isVideo) // 接收到的是视频格式
            {
                // 完成播放远程视频的Filter Graph的构建
                CompleteVideoReceivingGraph();
            }
            else   // 接收到的是音频格式
            {
                // 完成播放远程音频的Filter Graph的构建
                CompleteAudioReceivingGraph();
            }
            return true;
        }
    }
    return CMsgReceiver::ReceiveMessage(inMessage, ioParam, ioParam2);
}
// 完成播放远程视频的Filter Graph的构建
void CRoleAdmin::CompleteVideoReceivingGraph(void)
{
    if (mRemoteVideoGraph && mVideoReceiver && mVideoRenderer)
    {
        IPin * pOut = mVideoReceiver->GetPin(0);
        IPin * pIn  = mVideoRenderer->GetPin(TRUE);
        // 使用智能连接，将接收视频数据的Net Receiver与Video Renderer相连
        HRESULT hr = mRemoteVideoGraph->GetGraph()->Connect(pOut, pIn);
        // 设置视频显示窗口
        if (mVideoWindow)
        {
            mVideoWindow->SetGraph(mRemoteVideoGraph);
        }
        // 运行这个Filter Graph
        mRemoteVideoGraph->Run();
    }
}
// 完成播放远程音频的Filter Graph的构建
void CRoleAdmin::CompleteAudioReceivingGraph(void)
{
    if (mRemoteAudioGraph && mAudioReceiver && mAudioRenderer)
    {
        IPin * pOut = mAudioReceiver->GetPin(0);
        IPin * pIn  = mAudioRenderer->GetPin(TRUE);
        // 使用智能连接，将接收音频数据的Net Receiver与Audio Renderer相连
        HRESULT hr = mRemoteAudioGraph->GetGraph()->Connect(pOut, pIn);
        // 运行这个Filter Graph
        mRemoteAudioGraph->Run();
    }
}
// 发送一个不带参数的简单命令
BOOL CRoleAdmin::SendSimpleCommand(long inCommand)
{
    UDP_Pack myPack;
```

```
    ZeroMemory(&myPack, sizeof(myPack));
    myPack.command = inCommand;      // 指定命令类型
    myPack.my_hton();                // 转换字节顺序
    return mUdpManager.Send((char*)&myPack, sizeof(myPack)); // 发送命令
}
// 将本地的采集设备配置信息发送给远程端
BOOL CRoleAdmin::SendLocalDeviceConfigToPeer(void)
{
    UDP_Pack myPack;
    ZeroMemory(&myPack, sizeof(myPack));
    myPack.command = cmd_DeviceConfig;  // 指定命令类型
    // 取得本地采集设备的配置情况（作为命令的第1个参数）
    myPack.param1  = (mDeviceConfig & Local_Device_Mask);
    myPack.my_hton(); // 转换字节顺序
    return mUdpManager.Send((char*)&myPack, sizeof(myPack)); // 发送命令
}
// 创建所有必要的Filter Graph（最多时需要创建4个，分别用于音/视频的发送/接收）
BOOL CRoleAdmin::CreateGraph(void)
{
    BOOL pass = TRUE;
    // 如果远程端将有视频发送过来，则创建一个播放远程视频的Filter Graph
    if ((mDeviceConfig & Remote_Has_Video) == Remote_Has_Video)
    {
        mRemoteVideoGraph = new CDXGraph();
        pass = mRemoteVideoGraph->Create();
    }
    // 如果远程端将有音频发送过来，则创建一个播放远程音频的Filter Graph
    if ((mDeviceConfig & Remote_Has_Audio) == Remote_Has_Audio)
    {
        mRemoteAudioGraph = new CDXGraph();
        pass = mRemoteAudioGraph->Create();
    }
    // 如果本地端将发送视频，则创建一个发送本地视频的Filter Graph
    if ((mDeviceConfig & Local_Has_Video) == Local_Has_Video)
    {
        mLocalVideoGraph = new CDXGraph();
        pass = mLocalVideoGraph->Create();
    }
    // 如果本地端将发送音频，则创建一个发送本地音频的Filter Graph
    if ((mDeviceConfig & Local_Has_Audio) == Local_Has_Audio)
    {
        mLocalAudioGraph = new CDXGraph();
        pass = mLocalAudioGraph->Create();
    }
    return pass;
}
// 销毁Filter Graph
void CRoleAdmin::DeleteGraph(void)
{
    // 将视频窗口与Filter Graph去掉关联
    if (mVideoWindow)
    {
```

```
        mVideoWindow->SetGraph(0);
    }
    // 销毁所有创建过的Filter Graph
    SAFE_DELETE(mRemoteVideoGraph);
    SAFE_DELETE(mRemoteAudioGraph);
    SAFE_DELETE(mLocalVideoGraph);
    SAFE_DELETE(mLocalAudioGraph);
}
// 创建输入部分的Filter
BOOL CRoleAdmin::CreateInputFilters(void)
{
    BOOL  pass = FALSE;
    HRESULT hr = NOERROR;
    // 如果远程端将有视频发送过来，则创建一个接收远程视频的Source Filter
    if ((mDeviceConfig & Remote_Has_Video) == Remote_Has_Video)
    {
        mVideoReceiver = new CFilterNetReceiver(NULL, &hr);
        IBaseFilter * pFilter = NULL;
        mVideoReceiver->QueryInterface(IID_IBaseFilter,
            (void**)&pFilter);
        pass |= mRemoteVideoGraph->AddFilter(pFilter, L"Video Receiver");
        // 监听来自网络视频接收Filter的消息
        mVideoReceiver->AddMsgReceiver(this);
    }
    // 如果远程端将有音频发送过来，则创建一个接收远程音频的Source Filter
    if ((mDeviceConfig & Remote_Has_Audio) == Remote_Has_Audio)
    {
        mAudioReceiver = new CFilterNetReceiver(NULL, &hr);
        IBaseFilter * pFilter = NULL;
        mAudioReceiver->QueryInterface(IID_IBaseFilter,
            (void**)&pFilter);
        pass |= mRemoteAudioGraph->AddFilter(pFilter, L"Audio Receiver");
        // 监听来自网络音频接收Filter的消息
        mAudioReceiver->AddMsgReceiver(this);
    }
    // 如果本地端将发送视频，则创建一个视频采集设备Filter
    if ((mDeviceConfig & Local_Has_Video) == Local_Has_Video)
    {
        mVideoFilter = new
            CVideoCaptureFilter2(mLocalVideoGraph->GetGraph());
        ((CVideoCaptureFilter2 *)mVideoFilter)->SetDevice(mVideoDevice);
        pass |= mVideoFilter->CreateFilter();
    }
    // 如果本地端将发送音频，则创建一个音频采集设备Filter
    if ((mDeviceConfig & Local_Has_Audio) == Local_Has_Audio)
    {
        mAudioFilter = new
            CAudioCaptureFilter2(mLocalAudioGraph->GetGraph());
        ((CAudioCaptureFilter2 *)mAudioFilter)->SetDevice(mAudioDevice);
        pass |= mAudioFilter->CreateFilter();
    }
    return pass;
```

```
}
// 销毁输入部分的Filter
void CRoleAdmin::ReleaseInputFilters(void)
{
    SAFE_RELEASE(mVideoReceiver);
    SAFE_RELEASE(mAudioReceiver);
    SAFE_DELETE(mVideoFilter);
    SAFE_DELETE(mAudioFilter);
}
// 创建输出部分的Filter
BOOL CRoleAdmin::CreateOutputFilters(void)
{
    BOOL  pass = FALSE;
    HRESULT hr = NOERROR;
    // 如果远程端将有视频发送过来，则创建一个Video Renderer用于显示视频
    if ((mDeviceConfig & Remote_Has_Video) == Remote_Has_Video)
    {
        mVideoRenderer = new CDXFilter(mRemoteVideoGraph->GetGraph(),
            CLSID_VideoRenderer, "Video Renderer");
        pass |= mVideoRenderer->CreateFilter();
    }
    // 如果远程端将有音频发送过来，则创建一个Audio Renderer用于播放声音
    if ((mDeviceConfig & Remote_Has_Audio) == Remote_Has_Audio)
    {
        mAudioRenderer = new CDXFilter(mRemoteAudioGraph->GetGraph(),
            CLSID_DSoundRender, "Audio Renderer");
        pass |= mAudioRenderer->CreateFilter();
    }
    // 如果本地端将发送视频，则创建一个用于发送本地视频的Net Sender Filter
    if ((mDeviceConfig & Local_Has_Video) == Local_Has_Video)
    {
        mVideoSender = new CFilterNetSender(NULL, &hr);
        IBaseFilter * pFilter = NULL;
        mVideoSender->QueryInterface(IID_IBaseFilter, (void**)&pFilter);
        pass |= mLocalVideoGraph->AddFilter(pFilter, L"Video Sender");
    }
    // 如果本地端将发送音频，则创建一个用于发送本地音频的Net Sender Filter
    if ((mDeviceConfig & Local_Has_Audio) == Local_Has_Audio)
    {
        mAudioSender = new CFilterNetSender(NULL, &hr);
        IBaseFilter * pFilter = NULL;
        mAudioSender->QueryInterface(IID_IBaseFilter, (void**)&pFilter);
        pass |= mLocalAudioGraph->AddFilter(pFilter, L"Audio Sender");
    }
    return pass;
}
// 销毁输出部分的Filter
void CRoleAdmin::ReleaseOutputFilters(void)
{
    SAFE_DELETE(mVideoRenderer);
    SAFE_DELETE(mAudioRenderer);
    SAFE_RELEASE(mVideoSender);
```

```
        SAFE_RELEASE(mAudioSender);
    }
    // 将输入部分的Filter与输出部分的Filter相连,
    // 注意：此时用于接收远程数据的Net Receiver尚未接收到格式数据，所以暂不连接
    BOOL CRoleAdmin::ConnectFilters(void)
    {
        BOOL   pass = TRUE;
        IPin * pOut = NULL;
        IPin * pIn  = NULL;
        // 将视频采集设备Filter与用于发送视频数据的Net Sender连接
        if (mLocalVideoGraph && mVideoFilter && mVideoSender)
        {
            if (mVideoDevice->IsDVDevice()) // 采集设备是数码摄像机
            {
                pOut = mVideoFilter->GetPin(FALSE, "DV Vid Out");
                pIn  = mVideoSender->GetPin(0);
                pass = RenderDVVideo(mLocalVideoGraph, pOut, pIn);
            }
            else // 采集设备是USB摄像头
            {
                pOut = mVideoFilter->GetPin(FALSE, "Capture");
                pIn  = mVideoSender->GetPin(0);
                pass = mLocalVideoGraph->ConnectFilters(pOut, pIn);
            }
        }
        // 将音频采集设备Filter与用于发送音频数据的Net Sender连接
        if (mLocalAudioGraph && mAudioFilter && mAudioSender)
        {
            pOut = mAudioFilter->GetPin(FALSE);
            pIn  = mAudioSender->GetPin(0);
            pass = mLocalAudioGraph->ConnectFilters(pOut, pIn);
        }
        return pass;
    }
    // 当完成Filter Graph的构建之后进行一些操作，如让接收网络数据的Filter启动接收线程，
    // 运行本地用于发送音视频数据的Filter Graph等
    BOOL CRoleAdmin::AfterBuildGraph(void)
    {
        // 将用于视频传输的Socket设置给接收网络视频数据的Net Receiver
        BOOL   socketUsed  = FALSE;
        SOCKET videoSocket = GetVideoStreamSocket();
        if (mVideoReceiver && videoSocket != INVALID_SOCKET)
        {
            // 播放远程视频的Filter Graph不使用参考时钟
            // 以最快速度播放远程端传送过来的视频图像
            mRemoteVideoGraph->SetSyncSource(NULL);
            mVideoReceiver->SetStreamSocket(videoSocket);
            // 开始接收远程端发送过来的数据
            mVideoReceiver->StartReceiving();
            socketUsed = TRUE;
        }
        // 将用于视频传输的Socket设置给发送本地视频数据的Net Sender
```

```
    if (mVideoSender && videoSocket != INVALID_SOCKET)
    {
        mVideoSender->SetStreamSocket(videoSocket);
        socketUsed = TRUE;
    }
    // 如果视频传输的Socket没有被使用，则将这条Socket连接关闭
    if (!socketUsed && videoSocket != INVALID_SOCKET)
    {
        closesocket(videoSocket);
    }

    // 将用于音频传输的Socket设置给接收网络音频数据的Net Receiver
    socketUsed  = FALSE;
    SOCKET audioSocket = GetAudioStreamSocket();
    if (mAudioReceiver && audioSocket != INVALID_SOCKET)
    {
        mAudioReceiver->SetStreamSocket(audioSocket);
        // 开始接收远程端发送过来的数据
        mAudioReceiver->StartReceiving();
        socketUsed = TRUE;
    }
    // 将用于音频传输的Socket设置给发送本地音频数据的Net Sender
    if (mAudioSender && audioSocket != INVALID_SOCKET)
    {
        mAudioSender->SetStreamSocket(audioSocket);
        socketUsed = TRUE;
    }
    // 如果音频传输的Socket没有被使用，则将这条Socket连接关闭
    if (!socketUsed && audioSocket != INVALID_SOCKET)
    {
        closesocket(audioSocket);
    }

    // 运行本地用于发送音视频数据的Filter Graph
    if (mLocalVideoGraph)
    {
        mLocalVideoGraph->Run();
    }
    if (mLocalAudioGraph)
    {
        mLocalAudioGraph->Run();
    }
    return TRUE;
}
// 得到用于视频传输的Socket（服务器和客户机必须定制！）
SOCKET CRoleAdmin::GetVideoStreamSocket(void)
{
    return INVALID_SOCKET;
}
// 得到用于音频传输的Socket（服务器和客户机必须定制！）
SOCKET CRoleAdmin::GetAudioStreamSocket(void)
{
```

```
    return INVALID_SOCKET;
}

//
// CServerAdmin.h
//

#ifndef __H_CServerAdmin__
#define __H_CServerAdmin__

#include "CRoleAdmin.h"
#include "CStreamSocket.h"

class CServerAdmin : public CRoleAdmin
{
private:
    CStreamSocket   mVideoStream;  // 用于视频传输的Socket
    CStreamSocket   mAudioStream;  // 用于音频传输的Socket
public:
    CServerAdmin();
    virtual ~CServerAdmin();
    // 处理自定义消息
    virtual bool ReceiveMessage(MessageT inMessage, void * ioParam, void *
        ioParam2);
protected:
    // 服务器程序定制的用于获取音视频传输用的Socket的函数
    virtual SOCKET GetVideoStreamSocket(void);
    virtual SOCKET GetAudioStreamSocket(void);
};

#endif // __H_CServerAdmin__

//
// CServerAdmin.cpp
//

#include "stdafx.h"
#include <streams.h>
#include "CServerAdmin.h"
#include "GlobalDefs.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//////////////////////////////////////////////////////////////////////////
CServerAdmin::CServerAdmin()
{
    // 对于服务器来说，服务器端口就是本地端口，客户机端口就是远程端口
    mLocalUDPPort  = SERVER_UDP_PORT;  // 10096
```

```
    mTargetUDPPort = CLIENT_UDP_PORT;  // 10098
}
CServerAdmin::~CServerAdmin()
{
}
// 得到用于视频传输的Socket
SOCKET CServerAdmin::GetVideoStreamSocket(void)
{
    return mVideoStream.GetSocket();
}
// 得到用于音频传输的Socket
SOCKET CServerAdmin::GetAudioStreamSocket(void)
{
    return mAudioStream.GetSocket();
}
// 处理自定义消息，如接收到的各种UDP命令
bool CServerAdmin::ReceiveMessage(MessageT inMessage, void * ioParam, void
    * ioParam2)
{
    switch (inMessage)
    {
    case msg_UDPCommandReceived:    // 接收到了一个UDP命令
        {
            UDP_Pack * pCmd = (UDP_Pack*) ioParam;
            pCmd->my_ntoh();          // 转换字节顺序
            switch (pCmd->command)  // 分析命令类型
            {
            case cmd_ClientCalling: // 客户机连接请求
                // 得到客户机的IP地址
                SetTargetIP(ntohl(inet_addr((char*)ioParam2)));
                // 将服务器的采集设备配置信息发送给客户机
                SendLocalDeviceConfigToPeer();
                break;

            case cmd_DeviceConfig:  // 接收到了对方的设备配置信息
                // 保存远程端的设备配置信息
                SetDeviceConfig(pCmd->param1 << 2);
                // 开始建立两条Socket连接（为传输媒体数据做准备）
                mVideoStream.ConnectTo(mTargetIP, CLIENT_TCP_PORT);
                mAudioStream.ConnectTo(mTargetIP, CLIENT_TCP_PORT);
                break;

            case cmd_BuildFilterGraph: // 可以开始构建Filter Graph了
                {
                    // 让上层监听者（即主界面）构建Filter Graph
                    BOOL isBuilding = TRUE;
                    Broadcast(msg_ModifyFilterGraph, &isBuilding);
                }
                break;

            case cmd_DisconnectRequest: // 要求断开连接
                {
```

```
                // 让上层监听者（即主界面）销毁Filter Graph
                BOOL isBuilding = FALSE;
                Broadcast(msg_ModifyFilterGraph, &isBuilding);
            }
            break;
        }
    }
    return true;
  }

  return CRoleAdmin::ReceiveMessage(inMessage, ioParam, ioParam2);
}

//
// CClientAdmin.h
//

#ifndef __H_CClientAdmin__
#define __H_CClientAdmin__

#include "CRoleAdmin.h"
#include "CTCPListener.h"

class CClientAdmin : public CRoleAdmin
{
private:
    CTCPListener    mTcpListener;   // TCP连接管理对象
public:
    CClientAdmin();
    virtual ~CClientAdmin();
    // 重写初始化和反初始化函数
    virtual BOOL Init(void);
    virtual void Uninit(void);
    // 真正实现呼叫服务器函数
    virtual BOOL CallServer(void);
    // 处理自定义消息
    virtual bool ReceiveMessage(MessageT inMessage, void * ioParam, void *
        ioParam2);
protected:
    // 客户机程序定制的用于获取音视频传输用的Socket的函数
    virtual SOCKET GetVideoStreamSocket(void);
    virtual SOCKET GetAudioStreamSocket(void);
};

#endif // __H_CClientAdmin__

//
// CClientAdmin.cpp
//

#include "stdafx.h"
#include <streams.h>
```

```
#include "CClientAdmin.h"
#include "GlobalDefs.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////////////////////////////////////////////////////
CClientAdmin::CClientAdmin()
{
    // 对于客户机来说，客户机端口就是本地端口，服务器端口就是远程端口
    mLocalUDPPort  = CLIENT_UDP_PORT;  // 10098
    mTargetUDPPort = SERVER_UDP_PORT;  // 10096
    // 监听来自TCP连接管理对象的消息（当建立新的Socket连接时）
    mTcpListener.AddMsgReceiver(this);
}
CClientAdmin::~CClientAdmin()
{
    Uninit();  // 反初始化
}
// 重写父类的初始化函数
BOOL CClientAdmin::Init(void)
{
    BOOL pass = CRoleAdmin::Init();
    // 创建用于侦听TCP连接的Socket
    if (pass)
    {
        mTcpListener.SetListenPort(CLIENT_TCP_PORT);  // 10099
        pass = mTcpListener.Create();
    }
    if (pass)
    {
        // 启动一个侦听子线程
        pass = mTcpListener.StartListening();
    }
    return pass;
}
// 重写父类的反初始化函数
void CClientAdmin::Uninit(void)
{
    // 停止TCP连接的侦听线程
    mTcpListener.StopListening();
    CRoleAdmin::Uninit();
}
// 真正实现客户机呼叫服务器
BOOL CClientAdmin::CallServer(void)
{
    // 向服务器发送一个cmd_ClientCalling命令
    return SendSimpleCommand(cmd_ClientCalling);
}
```

```
// 得到用于视频传输的Socket（保存在TCP连接管理对象中）
SOCKET CClientAdmin::GetVideoStreamSocket(void)
{
    return mTcpListener.GetAccepted(0);
}
// 得到用于音频传输的Socket（保存在TCP连接管理对象中）
SOCKET CClientAdmin::GetAudioStreamSocket(void)
{
    return mTcpListener.GetAccepted(1);
}
// 处理自定义消息，如接收到的各种UDP命令
bool CClientAdmin::ReceiveMessage(MessageT inMessage, void * ioParam, void
    * ioParam2)
{
    switch (inMessage)
    {
    case msg_UDPCommandReceived:    // 接收到了一个UDP命令
        {
            UDP_Pack * pCmd = (UDP_Pack*) ioParam;
            pCmd->my_ntoh();          // 转换字节顺序
            switch (pCmd->command)  // 分析命令类型
            {
            case cmd_DeviceConfig:  // 接收到了对方的设备配置信息
                // 保存远程端的设备配置信息
                SetDeviceConfig(pCmd->param1 << 2);
                // 同时将客户机的设备配置信息发送给服务器
                SendLocalDeviceConfigToPeer();
                break;

            case cmd_DisconnectRequest: // 要求断开连接
                {
                    // 让上层监听者（即主界面）销毁Filter Graph
                    BOOL isBuilding = FALSE;
                    Broadcast(msg_ModifyFilterGraph, &isBuilding);
                }
                break;
            }
        }
        return true;

    case msg_TCPSocketAccepted: // 新的Socket连接已经建立的通知
        // 判断用于音频和视频传输的2条Socket连接是否都已经建立
        if (mTcpListener.IsSocketReady())
        {
            // 发送命令，要求服务器构建它自己的Filter Graph
            SendSimpleCommand(cmd_BuildFilterGraph);
            // 让上层监听者（即主界面）构建客户机的Filter Graph
            BOOL isBuilding = TRUE;
            Broadcast(msg_ModifyFilterGraph, &isBuilding);
        }
        return true;
    }
```

```
    return CRoleAdmin::ReceiveMessage(inMessage, ioParam, ioParam2);
}
```

从上述的实现中可以看出，服务器程序的Filter Graph是得到客户机的通知后才开始构建的。服务器和客户机双方的Filter Graph，都是通过向控制对象的上层监听者（即主界面）发送一个msg_ModifyFilterGraph消息来完成的。这个消息的处理过程如下：

```
// AVChatDlg.cpp : implementation file
//
#define WM_ModifyFilterGraph        (WM_USER+123)

BEGIN_MESSAGE_MAP(CAVChatDlg, CDialog)
    //{{AFX_MSG_MAP(CAVChatDlg)
    ON_MESSAGE(WM_ModifyFilterGraph, OnModifyFilterGraph)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

// 自定义消息处理函数（监听来自控制对象的消息）
bool CAVChatDlg::ReceiveMessage(MessageT inMessage, void * ioParam, void *
    ioParam2)
{
    if (inMessage == msg_ModifyFilterGraph)
    {
        // 在主线程的消息队列中放一个WM_ModifyFilterGraph消息
        PostMessage(WM_ModifyFilterGraph, *((BOOL*)ioParam));
        return true;
    }
    return CMsgReceiver::ReceiveMessage(inMessage, ioParam, ioParam2);
}
// WM_ModifyFilterGraph消息的处理函数
LRESULT CAVChatDlg::OnModifyFilterGraph(WPARAM inWParam, LPARAM inLParam)
{
    if (mActor)
    {
        if (inWParam)  // 是创建还是销毁
        {
            // 创建Filter Graph
            mActor->Activate();
        }
        else
        {
            // 销毁Filter Graph
            mActor->Deactivate();
        }
    }
    return 0;
}
```

读者可能感到奇怪，上述创建Filter Graph的过程实际上绕了一个圈子，最终还是要调用控制对象的Activate或Deactivate函数。那为什么不在控制类中接收到UDP命令后，直接（在UDP数据接收线程中）调用Activate或Deactivate函数呢？原因有两个。一是主线程有

消息循环，而UDP数据接收线程没有消息循环。AVChat程序使用CoCreateInstance函数显式地创建Video Renderer，这种情况下要求创建它的线程实现消息循环，否则视频窗口的消息将得不到处理。二是调用COM库函数（比如CoCreateInstance等）的每个线程都要进行COM库的初始化，否则这些函数调用将会失败。主线程在程序启动的时候就进行了COM库的初始化，而在UDP数据接收线程中没有这么做。当然，也可以给UDP数据接收线程加上COM库的初始化操作，但将所有COM组件放到同一个线程中去创建，程序本身更容易实现管理和控制。

最后来看一下，AVChat程序是如何使用CServerAdmin类和CClientAdmin类来完成角色控制的。首先，AVChat程序定义了一个枚举类型ROLE来区分角色。

```
typedef enum
{
    SERVER = 0,      // 服务器
    CLIENT           // 客户机
} ROLE;
```

AVChat程序到底扮演哪种角色，是通过弹出角色选择对话框让用户来确定的。角色信息保存在主界面类CAVChatDlg的一个ROLE类型的成员mRole中。然后根据mRole的不同取值，实例化不同的角色控制对象。这个过程实现参见CAVChatDlg类的Initialize函数。

```
// AVChatDlg.cpp : implementation file
//
void CAVChatDlg::Initialize(void)
{
    // 弹出角色选择对话框
    CRoleDlg  roleDlg;
    roleDlg.DoModal();
    // 销毁以前使用的角色控制对象
    SAFE_DELETE(mActor);
    // 获取用户选择的角色
    mRole = roleDlg.GetRole();
    if (mRole == SERVER)
    {
        // 如果角色是服务器，则实例化CServerAdmin类
        mActor = new CServerAdmin();
    }
    else
    {
        // 如果角色是客户机，则实例化CClientAdmin类
        mActor = new CClientAdmin();
    }
    // 主界面监听来自控制对象的消息
    mActor->AddMsgReceiver(this);
    // 设置视频显示窗口
    mActor->SetVideoWindow(&mVideoWindow);
    // 设置保存本地采集设备信息的对象指针
    mActor->SetAVDevice(&mVideoDevice, &mAudioDevice);
    // 控制对象进行初始化
    mActor->Init();
```

```
    // 弹出自检对话框进行本地采集设备的自检
    CSelfCheckDlg chkDlg;
    chkDlg.SetAVDevice(&mVideoDevice, &mAudioDevice);
    chkDlg.DoModal();
    // 获取本地采集设备配置情况，然后设置给控制对象，控制对象是根据设备配置情况来决定
    // 是否要创建特定的Filter Graph的
    mActor->SetDeviceConfig(chkDlg.GetDeviceConfig());
    // 刷新界面状态
    ReflectStatus();
}
// 主界面上Call按钮的响应函数
void CAVChatDlg::OnButtonCall()
{
    // 弹出对话框让用户指定服务器的IP地址
    CCallDlg  callDlg;
    if (callDlg.DoModal() == IDOK)
    {
        // 角色验证：仅支持客户机呼叫服务器
        if (mRole == CLIENT && mActor)
        {
            // 设置远程端（即服务器）的IP地址
            mActor->SetTargetIP(callDlg.GetServerIP());
            // 开始呼叫过程……
            mActor->CallServer();
            // 刷新界面上的按钮状态
            mButtonCall.EnableWindow(FALSE);
            mButtonDisconnect.EnableWindow(TRUE);
        }
    }
}
// 主界面上Disconnect按钮的响应函数
void CAVChatDlg::OnButtonDisconnect()
{
    if (mActor)
    {
        // 断开通话连接
        mActor->Disconnect();
        // 刷新界面上的按钮状态
        mButtonCall.EnableWindow(mRole == CLIENT);
        mButtonDisconnect.EnableWindow(FALSE);
        // 重新初始化控制对象，准备下一次的通话连接
        mActor->Init();
    }
}
```

3.2.5 开发网络发送Filter

在DirectShow中，Filter之间的数据传送以Sample的形式来完成。对于接入链路的任何一个Filter来说，都有机会获得数据流中的每一个Sample。因此，AVChat程序需要开发一个Filter接到采集设备Filter的后面，以获取采集输出的数据，然后才能向网络远程端发送。这

个Filter就是图3.15中的Net Sender。

很显然，Net Sender是一个Renderer Filter（因为它没有输出Pin）。因此，Filter的父类可以选择CBaseFilter类，输入Pin的父类选择CRenderedInputPin类。又因为Net Sender只在AVChat程序中使用，因此可以将它写成一种应用程序内的Filter形式。

先来看Net Sender的输入Pin的实现：

```
//
// CXInputPin.h
//

#ifndef __H_CMemDestInputPin__
#define __H_CMemDestInputPin__

#include "GlobalDefs.h"

class CFilterNetSender;
class CXInputPin : public CRenderedInputPin
{
    friend class CFilterNetSender;
private:
    CFilterNetSender * mFilter;          // Filter对象指针
    CCritSec         mReceiveLock;       // 用于数据接收同步
    SOCKET           mSocket;            // 用于发送数据的Socket（外部设入）
    Pack_Header      mPackHeader;        // 网络发送的数据包头
    BOOL             mIsMediaTypeSent;   // 媒体类型是否已经发送的标记
public:
    CXInputPin(CFilterNetSender *pFilter, HRESULT *phr);
    ~CXInputPin();
    // Filter上“委托”过来的函数
    void SetStreamSocket(SOCKET inSocket);
    void CancelPendingSend(void);
    // 重写基类Pin中的几个函数
    STDMETHODIMP Receive(IMediaSample *pSample);
    STDMETHODIMP EndOfStream(void);
    STDMETHODIMP BeginFlush(void);
    STDMETHODIMP EndFlush(void);
    STDMETHODIMP ReceiveCanBlock(void);
    // 定制Pin连接过程中的媒体类型检查
    HRESULT CheckMediaType(const CMediaType * inMediaType);
private:
    // 向远程端发送一次媒体格式信息
    int SendMediaType(void);
};

#endif // __H_CMemDestInputPin__

//
// CXInputPin.cpp
//

#include <streams.h>
```

```
#include <dvdmedia.h>
#include "CFilterNetSender.h"
#include "CXInputPin.h"
#include "stdio.h"

//////////////////////////////////////////////////////////////////////////////
CXInputPin::CXInputPin(CFilterNetSender *pFilter, HRESULT *phr) :
CRenderedInputPin(NAME("Input Pin"), pFilter, &pFilter->mFilterLock, phr,
L"Input")
{
    mFilter             = pFilter;
    mSocket             = INVALID_SOCKET;
    mIsMediaTypeSent    = FALSE;
}
CXInputPin::~CXInputPin()
{
    // 关闭Socket连接
    if (mSocket != INVALID_SOCKET)
    {
        closesocket(mSocket);
        mSocket = INVALID_SOCKET;
    }
}
// 从上一级Filter接收Sample数据，直接在DirectShow的数据流线程中进行网络数据的发送
STDMETHODIMP CXInputPin::Receive(IMediaSample *pSample)
{
    CAutoLock lck(&mReceiveLock);
    HRESULT hr = CRenderedInputPin::Receive(pSample);
    if (SUCCEEDED(hr))
    {
        // 判断Sample装的是否是媒体数据
        if (m_SampleProps.dwStreamId == AM_STREAM_MEDIA)
        {
            PBYTE pData;
            // 从Sample中得到数据指针，以及有效数据长度
            pSample->GetPointer(&pData);
            long length = pSample->GetActualDataLength();
            int bytes = 0;
            if (mSocket != INVALID_SOCKET)
            {
                // 如果还没有发送媒体格式信息，则先发送一次
                if (!mIsMediaTypeSent)
                {
                    mIsMediaTypeSent = TRUE;
                    bytes = SendMediaType();
                }
                // 开始发送Sample数据，也就是音视频数据
                // 首先发送一个包头
                mPackHeader.pack_type = PT_Payload;     // 负载类型
                mPackHeader.pack_size = length;         // 数据长度
                mPackHeader.my_hton();                  // 字节顺序转换
                bytes = send(mSocket, (char*)&mPackHeader,
```

```
                    sizeof (mPackHeader), 0);
                // 然后发送Sample中的媒体数据
                bytes = send(mSocket, (char*)pData, length, 0);
            }
        }
    }
    return hr;
}
// 向远程端发送一次媒体格式信息
int CXInputPin::SendMediaType(void)
{
    int bytes = 0;
    if (mSocket != INVALID_SOCKET)
    {
        // 判断输入Pin连接的媒体类型是视频还是音频
        if (m_mt.formattype == FORMAT_VideoInfo)
        {
            mPackHeader.pack_type = PT_VideoMediaType;
        }
        else
        {
            mPackHeader.pack_type = PT_AudioMediaType;
        }
        mPackHeader.pack_size = m_mt.cbFormat;      // 格式数据的长度
        mPackHeader.my_hton();                      // 字节顺序转换
        // 首先发送一个包头
        bytes = send(mSocket, (char*)&mPackHeader, sizeof(mPackHeader), 0);
        // 然后发送具体的媒体格式数据
        bytes = send(mSocket, (char *)m_mt.pbFormat, m_mt.cbFormat, 0);
    }
    return bytes;
}
// 数据流结束
STDMETHODIMP CXInputPin::EndOfStream(void)
{
    CAutoLock  lck(&mReceiveLock);
    return CRenderedInputPin::EndOfStream();
}
// 开始“清场”
STDMETHODIMP CXInputPin::BeginFlush(void)
{
    CAutoLock lck(&mFilter->mFilterLock);
    HRESULT hr = CRenderedInputPin::BeginFlush();
    return hr;
}
// 结束“清场”
STDMETHODIMP CXInputPin::EndFlush(void)
{
    CAutoLock lck(&mFilter->mFilterLock);
    return CRenderedInputPin::EndFlush();
}
// 询问用于接收数据的Receive函数会不会阻塞
```

```
STDMETHODIMP CXInputPin::ReceiveCanBlock(void)
{
    return S_OK;  // 可能阻塞
}
// 输入Pin连接过程中的媒体类型检查
HRESULT CXInputPin::CheckMediaType(const CMediaType * inMediaType)
{
    if (inMediaType->formattype == FORMAT_VideoInfo ||
        inMediaType->formattype == FORMAT_WaveFormatEx)
    {
        return S_OK;
    }
    return E_FAIL;
}
// 外部设置进来一个连接完好的Socket（用于数据发送）
void CXInputPin::SetStreamSocket(SOCKET inSocket)
{
    mSocket = inSocket;
}
// 放弃所有的阻塞操作
void CXInputPin::CancelPendingSend(void)
{
    // 关闭Socket连接以结束Socket的所有阻塞操作
    if (mSocket != INVALID_SOCKET)
    {
        closesocket(mSocket);
        mSocket = INVALID_SOCKET;
    }
}
```

然后再来看Net Sender Filter的实现：

```
//
// CFilterNetSender.h
//

#ifndef __H_CFilterNetSender__
#define __H_CFilterNetSender__

#include "CXInputPin.h"

class CFilterNetSender : public CBaseFilter
{
    friend class CXInputPin;
protected:
    CCritSec          mFilterLock;    // Filter对象上的同步
    CXInputPin *      mInputPin;      // 输入Pin指针
    CPosPassThru *    mPosition;      // 用于支持媒体定位
public:
    CFilterNetSender(LPUNKNOWN lpunk, HRESULT *phr);
    ~CFilterNetSender();
    DECLARE_IUNKNOWN;
    // 用于暴露本Filter上实现的接口
```

```
    STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void ** ppv);
    // 用于Pin枚举的函数
    CBasePin * GetPin(int n);
    int GetPinCount();
    // Filter进入停止状态
    STDMETHODIMP Stop();
    // 公共函数接口：设置连接好的Socket用于数据传送
    void SetStreamSocket(SOCKET inSocket);
};

#endif // __H_CFilterNetSender__

//
// CFilterNetSender.cpp
//

#include <streams.h>     // quartz, includes windows
#include <initguid.h>
#include "CFilterNetSender.h"

// 为Net Sender产生一个CLSID，无实际用处
// {A7F18A36-2899-4546-8D88-4D21D4F2DA4D}
DEFINE_GUID(CLSID_NetSender,
0xa7f18a36, 0x2899, 0x4546, 0x8d, 0x88, 0x4d, 0x21, 0xd4, 0xf2, 0xda, 0x4d);

// ---------------------------------------------------------------------
//          Filter implementation
// ---------------------------------------------------------------------
CFilterNetSender::CFilterNetSender(LPUNKNOWN lpunk, HRESULT *phr) :
CBaseFilter(NAME("NetSender"), lpunk, &mFilterLock, CLSID_NetSender)
{
    mPosition = NULL;
    // 创建一个输入Pin对象
    mInputPin = new CXInputPin(this, phr);
}
CFilterNetSender::~CFilterNetSender()
{
    // 记得将输入Pin对象销毁
    delete mInputPin;
    if (mPosition != NULL)
    {
        delete mPosition;
        mPosition = NULL;
    }
}
// 暴露IMediaPosition和IMediaSeeking接口，用于支持媒体定位
// （这只是Renderer Filter的一般做法，对于实时源来说没有任何意义）
STDMETHODIMP CFilterNetSender::NonDelegatingQueryInterface(REFIID riid,
    void ** ppv)
{
    if (riid == IID_IMediaPosition || riid == IID_IMediaSeeking)
    {
```

```
        HRESULT hr = S_OK;
        if (mPosition == NULL)
        {
            mPosition  = new CPosPassThru(NAME("Pass Through"),(IUnknown *)
                GetOwner(), (HRESULT *) &hr, mInputPin);
            if (mPosition == NULL)
            {
                return E_OUTOFMEMORY;
            }
        }
        if (FAILED(hr))
        {
            delete mPosition;
            mPosition = NULL;
            return hr;
        }
        return mPosition->NonDelegatingQueryInterface(riid, ppv);
    }
    else
    {
        return CBaseFilter::NonDelegatingQueryInterface(riid, ppv);
    }
}
// 根据索引，返回Filter上的所有Pin对象指针
CBasePin * CFilterNetSender::GetPin(int n)
{
    if (n == 0)
    {
        return mInputPin;
    }
    else
    {
        return NULL;
    }
}
// 返回Filter上Pin的总数
int CFilterNetSender::GetPinCount()
{
    // 本Filter只有一个Pin，所以返回1
    return 1;
}
// Filter进入停止状态
STDMETHODIMP CFilterNetSender::Stop()
{
    // 放弃输入Pin上的所有阻塞操作，否则可能导致Filter无法实现状态转换
    if (mInputPin)
    {
        mInputPin->CancelPendingSend();
    }
    return CBaseFilter::Stop();
}
// 公共函数接口：设置连接好的Socket用于数据传送
```

```
void CFilterNetSender::SetStreamSocket(SOCKET inSocket)
{
    // 将这个Socket设置给输入Pin对象
    if (mInputPin)
    {
        mInputPin->SetStreamSocket(inSocket);
    }
}
```

应该说Net Sender的实现还是比较简单的。另外，因为Net Sender不是一个标准的Filter（一种标准的COM组件），因此Net Sender不再需要使用g_cTemplates模板填写注册信息，也不再需要实现DllRegisterServer和DllUnregisterServer这两个注册函数。只是还有一点需要提醒读者，就是在创建这种应用程序内的Filter时，不能再使用API函数CoCreateInstance（既然它不是一个标准的COM组件），而应该在应用程序中直接使用new操作来创建一个Filter类的实例。以下演示代码摘自AVChat程序：

```
//
// CRoleAdmin.cpp
//
// 创建输出部分的Filter
BOOL CRoleAdmin::CreateOutputFilters(void)
{
    // ……

    if (……)
    {
        // new出一个Filter类的实例
        mVideoSender = new CFilterNetSender(NULL, &hr);
        // 获得IBaseFilter接口以增加一个引用计数，主要是防止Filter对象的意外自销毁
        IBaseFilter * pFilter = NULL;
        mVideoSender->QueryInterface(IID_IBaseFilter, (void**)&pFilter);
        // 将Net Sender加入到Filter Graph中
        pass |= mLocalVideoGraph->AddFilter(pFilter, L"Video Sender");
    }

    // ……
}
// 销毁输出部分的Filter
void CRoleAdmin::ReleaseOutputFilters(void)
{
    // 仍然像一般的COM接口一样释放！
    SAFE_RELEASE(mVideoSender);
    // ……
}
```

3.2.6 开发网络接收Filter

参与到DirectShow中工作的基本形式是Filter。因此，AVChat程序需要开发一个Filter用于接收网络数据，然后再将接收到的数据从该Filter的输出Pin上传送给下一级Filter。这

个用于网络接收的Filter就是图3.16中的Net Receiver。

Net Receiver和Net Sender的实现有些类似，都是一种应用程序内的Filter形式。不过，Net Receiver的具体实现要比Net Sender复杂一点：Net Receiver需要使用独立的线程进行网络数据的接收；当接收到第一个媒体格式数据后，必须马上通知上层应用程序完成Filter Graph的构建；然后继续接收媒体数据，并将这些数据以Sample的形式“推”给下一级Filter。

Net Receiver的Filter父类选择CBaseFilter类，输出Pin的父类选择CBaseOutputPin类。下面，先来看Net Receiver的输出Pin的实现：

```
//
// CNetOutPin.h
//

#ifndef __H_CNetOutPin__
#define __H_CNetOutPin__

class CFilterNetReceiver;
class CNetOutPin : public CBaseOutputPin
{
private:
    CFilterNetReceiver *    mFilter;        // Filter对象指针
    CMediaType              mPreferredMt;   // 输出Pin上使用的媒体类型
public:
    CNetOutPin(CFilterNetReceiver *inFilter, HRESULT *phr, LPCWSTR pName);
    ~CNetOutPin();
    // 根据格式数据重建输出Pin上使用的媒体类型
    void SetupMediaType(long inType, char * inFormat, long inLength);
    // 输出Pin连接过程中的媒体类型检查
    virtual HRESULT CheckMediaType(const CMediaType *);
    // 决定输出Pin上的Sample属性，如Sample内存大小、Sample个数等
    virtual HRESULT DecideBufferSize(IMemAllocator * pAlloc,
        ALLOCATOR_PROPERTIES * pprop);
    // 得到输出Pin上推荐使用的媒体类型
    HRESULT GetMediaType(int iPosition,CMediaType *pMediaType);
};

#endif // __H_CNetOutPin__

//
// CNetOutPin.cpp
//

#include "stdafx.h"
#include "GlobalDefs.h"
#include <streams.h>
#include "CFilterNetReceiver.h"
#include "CNetOutPin.h"

/////////////////////////////////////////////////////////////////////////////
CNetOutPin::CNetOutPin(CFilterNetReceiver *inFilter, HRESULT *phr, LPCWSTR
    pName) :
```

```
CBaseOutputPin(NAME("Net_Stream"), inFilter, &inFilter->mFilterLock, phr,
    pName)
{
    mFilter = inFilter;
    mPreferredMt.InitMediaType();
}
CNetOutPin::~CNetOutPin()
{
}
// 根据格式数据重建输出Pin上使用的媒体类型（当Socket上接收到格式数据后会调用该函数）
void CNetOutPin::SetupMediaType(long inType, char * inFormat, long inLength)
{
    if (inType == PT_VideoMediaType) // 视频格式
    {
        mPreferredMt.SetType(&MEDIATYPE_Video);
        mPreferredMt.SetFormatType(&FORMAT_VideoInfo);
        // 判断得出媒体类型的子类型
        VIDEOINFOHEADER * pvi = (VIDEOINFOHEADER *) inFormat;
        const GUID subtype = GetBitmapSubtype(&pvi->bmiHeader);
        mPreferredMt.SetSubtype(&subtype);
        // 修正图像帧大小
        pvi->bmiHeader.biSizeImage = pvi->bmiHeader.biWidth *
            pvi->bmiHeader.biHeight * pvi->bmiHeader.biBitCount / 8;
        // 设置格式数据
        mPreferredMt.SetFormat((BYTE*)inFormat, inLength);
        mPreferredMt.SetSampleSize(pvi->bmiHeader.biSizeImage);
    }
    else  // 音频格式
    {
        mPreferredMt.SetType(&MEDIATYPE_Audio);
        mPreferredMt.SetSubtype(&MEDIASUBTYPE_PCM);
        mPreferredMt.SetFormatType(&FORMAT_WaveFormatEx);
        // 设置格式数据
        WAVEFORMATEX * wave = (WAVEFORMATEX *) inFormat;
        wave->nAvgBytesPerSec = wave->nSamplesPerSec * wave->nChannels
            * wave->wBitsPerSample / 8;
        mPreferredMt.SetFormat((BYTE*)inFormat, inLength);
        mFilter->SetAudioBytesPerSecond(wave->nAvgBytesPerSec);
    }
    mPreferredMt.SetTemporalCompression(FALSE);
}
// 输出Pin连接过程中的媒体类型检查
HRESULT CNetOutPin::CheckMediaType(const CMediaType * inMediaType)
{
    if (*inMediaType == mPreferredMt)
    {
        return NOERROR;
    }
    return E_FAIL;
}
// 决定输出Pin上的Sample属性
HRESULT CNetOutPin::DecideBufferSize(IMemAllocator * pAlloc,
```

```
        ALLOCATOR_PROPERTIES * pprop)
{
    ASSERT(pAlloc);
    ASSERT(pprop);
    HRESULT hr = NOERROR;

    // 决定Sample的内存大小
    if (mPreferredMt.formattype == FORMAT_VideoInfo)
    {
        // 如果是视频数据，则Sample内存大小取值为图像帧大小
        VIDEOINFOHEADER * info = (VIDEOINFOHEADER *) mPreferredMt.pbFormat;
        pprop->cbBuffer = info->bmiHeader.biSizeImage;
    }
    else
    {
        // 如果是音频数据，则Sample内存大小取值为1秒钟的数据量大小
        WAVEFORMATEX * info = (WAVEFORMATEX *) mPreferredMt.pbFormat;
        pprop->cbBuffer = info->nAvgBytesPerSec;
    }
    pprop->cBuffers  = 1; // 使用1个Sample
    pprop->cbAlign   = 1; // Sample数据按1字节对齐

    ASSERT(pprop->cbBuffer);
    ALLOCATOR_PROPERTIES Actual;
    hr = pAlloc->SetProperties(pprop, &Actual);
    if (FAILED(hr))
    {
        return hr;
    }

    ASSERT( Actual.cBuffers == 1 );
    if (pprop->cBuffers > Actual.cBuffers ||
        pprop->cbBuffer > Actual.cbBuffer)
    {
        return E_FAIL;
    }
    return NOERROR;
}
// 得到输出Pin上推荐使用的媒体类型
HRESULT CNetOutPin::GetMediaType(int iPosition,CMediaType *pMediaType)
{
    if (iPosition == 0)
    {
        *pMediaType = mPreferredMt;
        return NOERROR;
    }
    return E_INVALIDARG;
}
```

然后再来看Net Receiver Filter的实现：

```
//
// CFilterNetReceiver.h
```

```
//

#ifndef __H_CFilterNetReceiver__
#define __H_CFilterNetReceiver__

#include "CNetOutPin.h"
#include "CStreamSocket.h"
#include "CMsgStation.h"

class CFilterNetReceiver : public CBaseFilter, public CMsgStation
{
    friend class CNetOutPin;
private:
    CCritSec            mFilterLock;        // Filter对象上的同步
    CNetOutPin *        mOutPin;            // 输出Pin对象指针
    BOOL                mIsVideo;           // 接收的是否是视频数据
    CStreamSocket       mReceiver;          // 用于接收网络数据的Socket包装类
    IMediaSample *      mSample;            // 当前用于传送的Sample
    long                mAudioBytesPerSecond;   // 音频每秒的数据量
    REFERENCE_TIME      mLastSampleTime;        // 上一个Sample上的结束时间
public:
    CFilterNetReceiver(LPUNKNOWN punk, HRESULT *phr);
    ~CFilterNetReceiver();
public:
    // 重写Filter基类的几个函数
    virtual int GetPinCount();
    virtual CBasePin * GetPin(int n);
    STDMETHODIMP Stop();
    // 设置音频每秒的数据量
    void SetAudioBytesPerSecond(long inBytes);
    // 得到输出Pin上的一个空闲Sample的数据内存地址
    BOOL GetSampleBuffer(PBYTE * outBuffer);
    // 将填好数据的Sample发送给下一级Filter
    BOOL DeliverHoldingSample(long inSampleSize);

    // --- 公共函数：主要供上层应用程序调用 ---
    // 根据从网络上接收到的格式数据重建输出Pin上使用的媒体类型
    void SetupMediaType(long inType, char * inFormat, long inLength);
    // 设置用于接收网络数据的Socket
    void SetStreamSocket(SOCKET inSocket);
    // 开始使用Socket接收网络数据
    void StartReceiving(void);
};

#endif // __H_CFilterNetReceiver__

//
// CFilterNetReceiver.cpp
//

#include "stdafx.h"
#include "GlobalDefs.h"
#include <streams.h>
#include "CFilterNetReceiver.h"
```

```
#include <initguid.h>

// 为Net Receiver产生一个CLSID，无实际用处
// {58E6FA74-AEEA-4129-A1C3-79107B535C6F}
DEFINE_GUID(CLSID_NetReceiver, 0x58e6fa74, 0xaeea, 0x4129, 0xa1, 0xc3,
    0x79, 0x10, 0x7b, 0x53, 0x5c, 0x6f);

///////////////////////////////////////////////////////////////////////////
CFilterNetReceiver::CFilterNetReceiver(LPUNKNOWN punk, HRESULT *phr) :
CBaseFilter(NAME("Reciever"), punk, &mFilterLock, CLSID_NetReceiver)
{
    // 创建一个输出Pin对象
    mOutPin  = new CNetOutPin(this, phr, L"Out");
    mIsVideo = TRUE;
    mSample  = NULL;
    mAudioBytesPerSecond = 0;
    mLastSampleTime      = 0;
    // 将Filter对象指针设置给Socket包装对象
    mReceiver.SetOwnerFilter(this);
}
CFilterNetReceiver::~CFilterNetReceiver()
{
    // 停止网络接收
    mReceiver.StopReceiving();
    // 如果输出Pin上的Sample还没有释放，则释放它
    if (mSample)
    {
        mSample->Release();
        mSample = NULL;
    }
    // 最后记得销毁输出Pin对象！
    delete mOutPin;
}
// 返回Filter上的Pin的数量
int CFilterNetReceiver::GetPinCount()
{
    return 1; // 本Filter只有1个Pin
}
// 根据索引返回Filter上的所有Pin
CBasePin * CFilterNetReceiver::GetPin(int n)
{
    if (n == 0)
    {
        return mOutPin;
    }
    else
    {
        return NULL;
    }
}
// Filter进入停止状态
STDMETHODIMP CFilterNetReceiver::Stop()
{
    CAutoLock lck(&mFilterLock);  // 获得同步控制权
```

```
    // 停止网络接收
    mReceiver.StopReceiving();
    // 释放输出Pin上没有送出去的Sample
    if (mSample)
    {
        mSample->Release();
        mSample = NULL;
    }
    return CBaseFilter::Stop();
}
// 设置音频每秒的数据量（在给音频Sample打时间戳的时候用于计算时间戳）
void CFilterNetReceiver::SetAudioBytesPerSecond(long inBytes)
{
    mAudioBytesPerSecond = inBytes;
}
// 得到输出Pin上的一个空闲Sample的数据内存地址
BOOL CFilterNetReceiver::GetSampleBuffer(PBYTE * outBuffer)
{
    // 将没有释放的Sample先释放掉
    if (mSample)
    {
        mSample->Release();
        mSample = NULL;
    }
    // 在输出Pin上得到一个新的空闲的Sample
    HRESULT hr = mOutPin->GetDeliveryBuffer(&mSample,NULL,NULL,0);
    if (mSample)
    {
        // 得到Sample的数据内存地址
        mSample->GetPointer(outBuffer);
    }
    return SUCCEEDED(hr);
}
// 将填好数据的Sample发送给下一级Filter
BOOL CFilterNetReceiver::DeliverHoldingSample(long inSampleSize)
{
    if (mSample)
    {
        // --- 设置Sample上的属性 ---
        mSample->SetActualDataLength(inSampleSize); // 有效数据长度
        mSample->SetSyncPoint(TRUE);  // 同步点
        if (mIsVideo)
        {
            // 如果是视频帧，不使用时间戳（以最快速度播放）
            mSample->SetTime(NULL, NULL);
        }
        else
        {
            // 为音频数据计算时间戳
            REFERENCE_TIME rtStart = mLastSampleTime;
            mLastSampleTime += (UNITS * inSampleSize /
                mAudioBytesPerSecond);
            REFERENCE_TIME rtEnd   = mLastSampleTime;
            // 给Sample打上时间戳
```

```
            mSample->SetTime(&rtStart, &rtEnd);
        }
        // 将Sample传送下去
        HRESULT hr = mOutPin->Deliver(mSample);
        // 释放Sample（让它回到空闲列表中去等候下一次使用）
        mSample->Release();
        mSample = NULL;
        return SUCCEEDED(hr);
    }
    return TRUE;
}
// 根据从网络上接收到的格式数据重建输出Pin上使用的媒体类型
void CFilterNetReceiver::SetupMediaType(long inType, char * inFormat, long
    inLength)
{
    // 判断将要接收的是视频数据还是音频数据
    if (inType == PT_VideoMediaType)
    {
        mIsVideo = TRUE;
    }
    else
    {
        mIsVideo = FALSE;
    }
    // 为输出Pin重建媒体类型
    mOutPin->SetupMediaType(inType, inFormat, inLength);
    // 最后通知上层的应用程序：是时候完成Filter Graph的最后构建啦！
    Broadcast(msg_MediaTypeReceived, &mIsVideo);
}
// 外部设置进来用于接收网络数据的Socket
void CFilterNetReceiver::SetStreamSocket(SOCKET inSocket)
{
    // 将Socket关联到包装对象中
    mReceiver.Attach(inSocket);
}
// 开始使用Socket接收网络数据
// （Socket包装对象内部使用了一个子线程专门用于数据接收）
void CFilterNetReceiver::StartReceiving(void)
{
    mReceiver.StartReceiving();
}
```

读者可能感到奇怪，“从上面的实现中，我并没有看到网络数据的接收过程啊！也没有看到Sample到底是怎么从输出Pin上传送出去的！”照理说，Filter或Pin上应该有一个线程，专门负责将从网络上接收到的数据以Sample的形式“推”出去。怎么回事呢？奥秘就在CFilterNetReceiver类使用的一个CStreamSocket类的实例上。这个CStreamSocket类似曾相识，但实际上它与3.1.1节中的CStreamSocket类有些不一样，主要的区别在于增加了一个接口函数设置使用它的Filter对象，还有就是接收过程的具体化（与Filter上的操作关联）。

```
//
// CStreamSocket.h
//
class CStreamSocket
```

```
{
protected:
    // 其他成员（省略）
    // ……
    CFilterNetReceiver * mOwnerFilter; // Filter对象指针
public:
    CStreamSocket();
    virtual ~CStreamSocket();
    // 设置使用本类的Filter对象指针
    void SetOwnerFilter(CFilterNetReceiver * inFilter);
    // 其他接口函数（省略）
    // ……
};

//
// CStreamSocket.cpp
//
// 设置使用本类的Filter对象指针
void CStreamSocket::SetOwnerFilter(CFilterNetReceiver * inFilter)
{
    mOwnerFilter = inFilter;
}
// 网络数据的循环接收过程
void CStreamSocket::ReceivingLoop(void)
{
    char   buf[1024];  // 接收缓存
    int    bytes = 0;
    Pack_Header * pHeader = (Pack_Header *) buf;
    PBYTE  pSampleBuffer;

    while (mIsReceiving && mOwnerFilter)
    {
        // 首先接收一个包头
        bytes = recv(mSocket, buf, sizeof(Pack_Header), 0);
        if (bytes == SOCKET_ERROR || bytes == 0)
        {
            Detach();
            mIsReceiving = FALSE;
            break;
        }

        pHeader->my_ntoh();  // 转换字节顺序
        // 如果包头后面的是音频或视频的格式数据
        if (pHeader->pack_type == PT_AudioMediaType ||
            pHeader->pack_type == PT_VideoMediaType)
        {
            // 接收包头后面的格式数据
            bytes = recv(mSocket,buf+sizeof(Pack_Header),pHeader->
                pack_size,0);
            if (bytes == SOCKET_ERROR || bytes == 0)
            {
                Detach();
                mIsReceiving = FALSE;
                break;
            }
```

```
            // 通知Filter重构媒体类型
            mOwnerFilter->SetupMediaType(pHeader->pack_type, buf + sizeof
                (Pack_Header), pHeader->pack_size);
        }
        else  // 接收的是媒体数据
        {
            BOOL pass = FALSE;
            do
            {
                // 得到一个输出Pin上的空闲Sample的内存地址
                pass = mOwnerFilter->GetSampleBuffer(&pSampleBuffer);
                if (!pass)
                {
                    // 如果得不到则等待100ms后再尝试
                    Sleep(100);
                }
            } while (mIsReceiving && !pass);

            long required = pHeader->pack_size; // 媒体数据长度
            long received = 0;
            // 循环接收数据，直到达到指定的长度，因为可能recv函数一次调用接收不到足
            // 够多的数据
            while (pass && mIsReceiving && required > 0)
            {
                bytes = recv(mSocket, (char*)(pSampleBuffer+received),
                    required, 0);
                if (bytes == SOCKET_ERROR || bytes == 0)
                {
                    Detach();
                    mIsReceiving = FALSE;
                    break;
                }
                received += bytes;  // 已经接收字节数
                required -= bytes;  // 还需接收字节数
            }
            // 将填充完毕的Sample传送给下一级Filter
            mOwnerFilter->DeliverHoldingSample(pHeader->pack_size);
        }
    }
}
```

因为Net Receiver与Net Sender一样，也是一种应用程序内的Filter形式，因此在这里还要强调一下它的使用方法。以下演示代码摘自AVChat程序：

```
//
// CRoleAdmin.cpp
//
// 创建输入部分的Filter
BOOL CRoleAdmin::CreateInputFilters(void)
{
    // ……

    if (……)
    {
        // new出一个Filter类的实例
```

```
        mVideoReceiver = new CFilterNetReceiver(NULL, &hr);
        // 获得IBaseFilter接口以增加一个引用计数，主要是防止Filter对象的意外自销毁
        IBaseFilter * pFilter = NULL;
        mVideoReceiver->QueryInterface(IID_IBaseFilter,
            (void**)&pFilter);
        // 将Net Receiver加入到Filter Graph中
        pass |= mRemoteVideoGraph->AddFilter(pFilter, L"Video Receiver");
        // 监听来自Net Receiver的自定义消息，如Net Receiver接收到了
        // 格式数据，告诉上层观察者是时候完成Filter Graph的构建了
        mVideoReceiver->AddMsgReceiver(this);
    }

    // ……
}
// 销毁输入部分的Filter
void CRoleAdmin::ReleaseInputFilters(void)
{
    // 仍然像一般的COM接口一样释放
    SAFE_RELEASE(mVideoReceiver);
    // ……
}
```

3.2.7 程序导读

首先来了解一下AVChat项目的VC工作区结构，如图3.19所示。

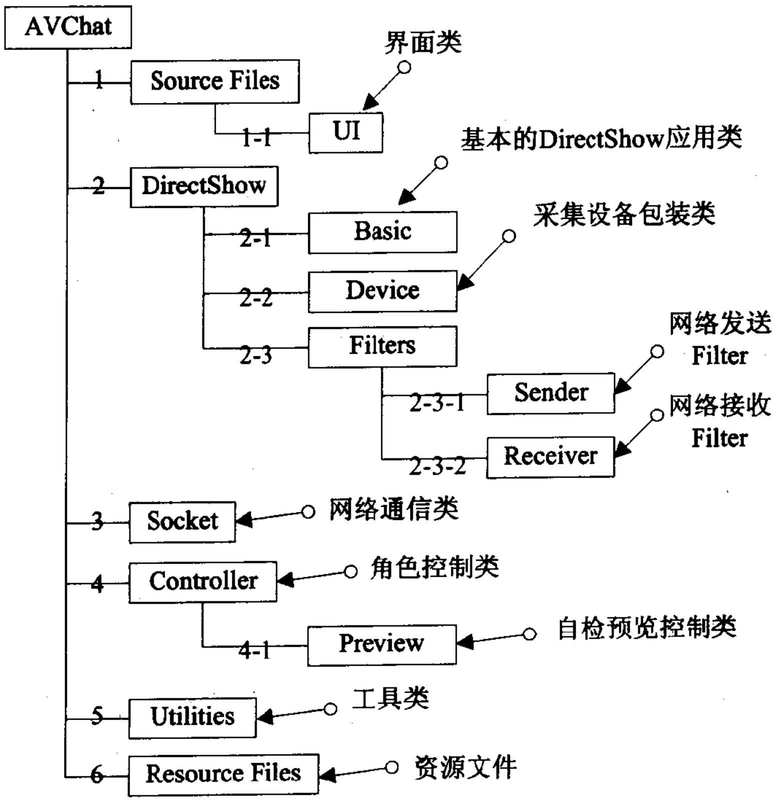


图 3.19　AVChat 项目的 VC 工作区结构

AVChat程序定义了很多类，这些类的清单如表3.2所示。

表3.2 AVChat程序中的所有类清单

类名	所在目录	功能描述
CAVChatDlg	1-1	主界面
CRoleDlg	1-1	角色选择对话框
CSelfCheckDlg	1-1	本地采集设备自检对话框
CCallDlg	1-1	呼叫服务器界面（用于指定服务器IP地址）
CDXGraph	2-1	封装了Filter Graph的控制功能
CDXFilter	2-1	Filter的包装类，并作为具体Filter包装类的父类
CVideoWindow	2-1	视频显示窗口（从CWnd类派生）
UDsUtils	2-1	工具类，完成了DirectShow相关的一系列相对独立的功能（比如创建设备Filter、显示Filter的属性页等）
UFilterUtils	2-1	工具类，完成了Filter相关的一系列相对独立的功能（比如得到某个Filter上的输入Pin、输出Pin、未连接的Pin、输入输出Pin的个数等）
CAVDevice	2	用于保存音视频采集设备信息（比如设备的显示名字、友好名字、设备类型等）
CAudioCaptureFilter2	2-2	音频采集设备的包装Filter，用于音频采集设备的创建、参数设置等
CVideoCaptureFilter2	2-2	视频采集设备的包装Filter，用于视频采集设备的创建、参数设置等
CFilterNetSender	2-3-1	用于向网络发送音视频数据的Filter（Net Sender）
CXInputPin	2-3-1	Net Sender的输入Pin
CFilterNetReceiver	2-3-2	用于接收来自网络的音视频数据的Filter（Net Receiver）
CNetOutPin	2-3-2	Net Receiver的输出Pin
CTCPListener	3	监听TCP连接的Socket包装类
CStreamSocket	3	使用TCP协议进行数据的发送和接收
CUDPManager	3	使用UDP协议进行数据的发送和接收
UNetwork	3	网络应用相关的工具类，用于获取主机的名字、IP地址等信息
CPreviewAdmin	4-1	用于本地采集设备自检的控制类，并作为角色控制类的父类，负责Filter Graph的构建、控制、销毁
CRoleAdmin	4	用于角色控制的基类，完成服务器和客户机两种角色控制逻辑中的公共部分
CServerAdmin	4	服务器进程使用的角色控制类
CClientAdmin	4	客户机进程使用的角色控制类
CMsgReceiver	5	自定义消息的接收者的基类
CMsgStation	5	自定义消息的发送者的基类
CObjectList	5	一个列表类（从CPtrArray类派生，列表元素为指针）

AVChat程序使用到的一些重要的类的继承结构如图3.20所示。

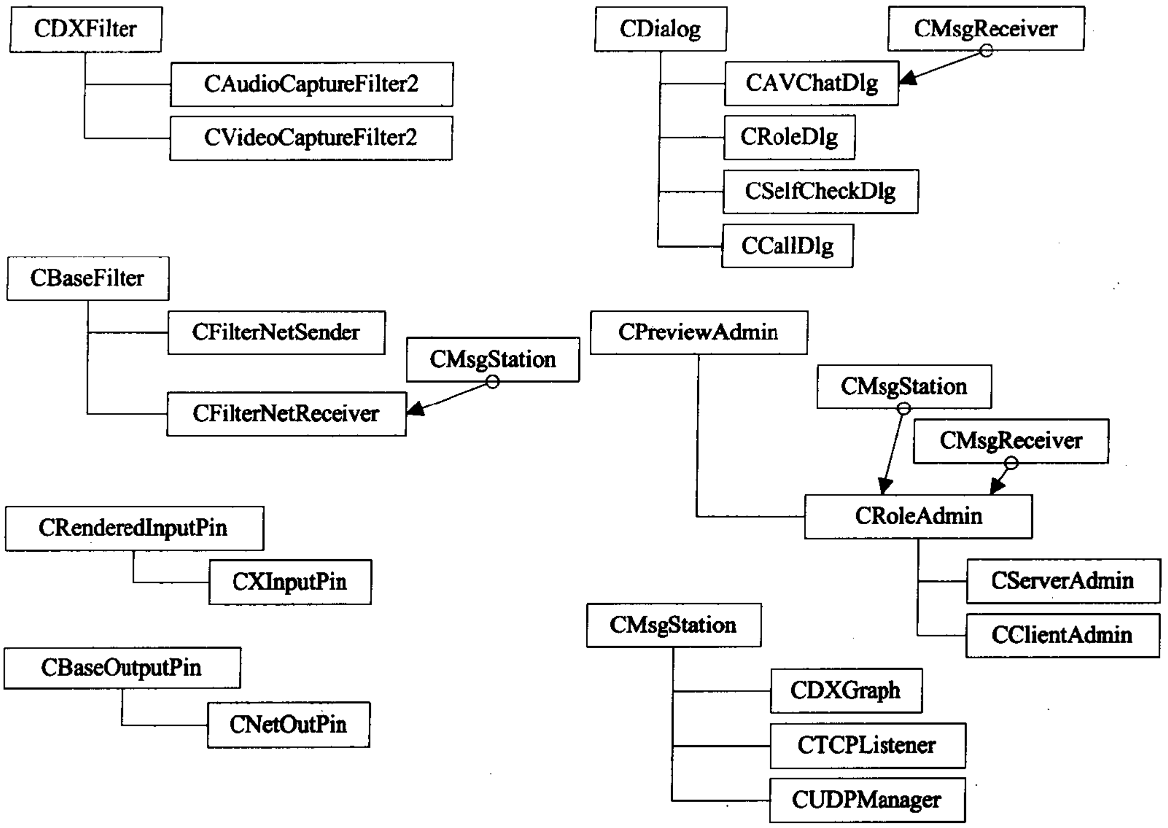


图 3.20 AVChat 程序的类继承结构

DirectShow应用程序是一种COM客户程序。因此AVChat程序应该在调用任何COM库函数（除了CoGetMalloc、CoTaskMemAlloc、CoTaskMemFree等几个内存操作相关的函数外）之前，完成COM库的初始化。如果有多个线程需要使用COM库函数，则每个线程都要进行这样的初始化。

COM库的初始化可以使用CoInitialize函数，也可以使用CoInitializeEx函数。在开发应用程序时，推荐使用CoInitializeEx，因为它包含了CoInitialize的功能。实际上，CoInitialize(NULL)等于CoInitializeEx（NULL，COINIT_APARTMENTTHREADED）。另外，COM库的反初始化使用CoUninitialize函数，它的调用必须与初始化函数一一对应。以下代码摘自AVChat程序。

```
// AVChat.cpp : Defines the class behaviors for the application.
//
BOOL CAVChatApp::InitInstance()
{
    // 初始化Socket库
    if (!AfxSocketInit())
    {
        AfxMessageBox("Socket initializing failded!");
        return FALSE;
    }
    //////////////////// 初始化COM库 ////////////////
    CoInitializeEx(NULL, COINIT_APARTMENTTHREADED);
    /////////////////////////////////////////////////
```

```
    // 创建并显示主对话框界面
    CAVChatDlg dlg;
    m_pMainWnd = &dlg;
    int nResponse = dlg.DoModal();
    if (nResponse == IDOK)
    {
        // TODO: Place code here to handle when the dialog is
        //  dismissed with OK
    }
    else if (nResponse == IDCANCEL)
    {
        // TODO: Place code here to handle when the dialog is
        //  dismissed with Cancel
    }
    return FALSE;
}
int CAVChatApp::ExitInstance()
{
    //////////// 反初始化COM库 /////////////
    CoUninitialize();
    ////////////////////////////////////////////
    return CWinApp::ExitInstance();
}
```

小知识：COM 的线程模型

CoInitializeEx函数的原型如下：

```
HRESULT CoInitializeEx(
      void * pvReserved,      // 保留，必须为NULL
      DWORD dwCoInit          // 指定线程模型
);
```

其中第二个参数用于指定COM的线程模型，它的取值为下列枚举类型COINIT中的一个。

```
typedef enum tagCOINIT {
COINIT_APARTMENTTHREADED              = 0x2,              // 单线程套间

#if (_WIN32_WINNT >= 0x0400 ) || defined(_WIN32_DCOM)  // DCOM
      COINIT_MULTITHREADED                  = 0x0,        // 多线程套间
      COINIT_DISABLE_OLE1DDE                = 0x4,
COINIT_SPEED_OVER_MEMORY                    = 0x8,
#endif // DCOM
} COINIT;
```

单线程套间STA（Single Threaded Apartment）——COM对象允许在多线程环境下使用，但是对象上的所有接口函数调用都在创建它的线程中完成；如果其他线程想要访问该对象，则必须通过向它的创建线程发送一个特定的窗口消息来完成（这种消息机制COM库已经帮我们实现好了）。通过这种方式，COM对象访问的并发性问题（多个线程同时访问对象资源引起冲突的问题）得到了解决。另外，使用STA套间的COM对象在实现时，代码上一般

就不用考虑并发性问题，因为对象上的接口函数调用都被外部使用的消息机制串行化了。惟一需要注意的是，STA套间线程必须实现一个消息循环。

多线程套间MTA（Multi-Threaded Apartment）也叫自由线程模型（Free-threading），在这种模型下，COM对象可以在一个线程中创建，而其他任何线程都可以直接调用这个对象上的接口函数。那么，如何来解决对象访问的并发性问题呢？这就要依赖COM对象自身的实现了。COM对象在实现时，代码上就要考虑并发访问的问题，比如使用关键区域（Critical Section）、信号量（Semaphore）、互斥量（Mutex）等多线程同步机制。

单线程套间使用消息机制解决了COM对象访问的并发性问题，非创建线程对COM对象的访问都要从创建线程的消息队列中“绕道”，这种工作方式是低效率的。而多线程套间通过COM对象本身来解决并发性问题，对COM对象的访问是直接的和高效率的，并且可以发挥多处理器的优点，实现跨线程、跨进程、跨机器的COM对象访问。

AVChat程序的主对话框在初始化时（CAVChatDlg::OnInitDialog函数中）启动了一个定时器，延后了控制对象的初始化（调用CAVChatDlg::Initialize函数）。那为什么不在OnInitDialog函数中直接调用Initialize函数，却要使用一个定时器呢？这么做的目的主要是让主对话框完全显示之后，再弹出角色选择对话框、设备自检对话框进行参数的调整，防止用户在上述两个弹出式对话框不慎失去焦点后找不到主对话框（此时主对话框的初始化函数被弹出式对话框阻塞住了，尚未显示出来，在Windows的任务栏上也将找不到AVChat程序的任何踪迹）。

AVChat程序的服务器进程和客户机进程的交互过程如图3.17所示，这里不再赘述。最后还有一点需要提醒读者的是，在使用AVChat程序进行视频聊天（兼带语音）时，可能会感到对方说话的声音有些滞后，这是什么原因造成的呢？大家知道，音频采集设备默认情况下使用的采集缓存能够容纳500ms的数据。只有采集满了500ms数据后，音频采集设备Filter才会送出一个Sample。而视频采集以帧率25fps为例，视频采集设备Filter每输出一个Sample（也就是一帧图像）的时间间隔是40ms。也就是说，音频数据要比视频数据迟送出来460ms。于是，在远程端接收到数据后进行播放时，音视频不同步也就不奇怪了。解决方法也很简单，只要在本地采集时减小音视频数据采集送出的时间差即可。也就是在CAudioCaptureFilter2类中成功创建了音频采集设备Filter之后，随即调用CaudioCapture Filter2::SetCaptureBufferSize函数进行音频采集缓存大小的调整。将缓存大小调小一点，但也不能太小，否则也会引起一些问题，有兴趣的读者可以继续深入研究一下。

提示：本书配套光盘的SourceCodes\Chapter03\AVChat目录下提供了视频聊天演示程序的完整实现。

3.3 视频点播

3.3.1 功能介绍

3.2节的视频聊天程序是融合了DirectShow技术与网络通信技术的典型应用。除此之外，

本书还要介绍另外一个DirectShow技术在网络方面应用的例子，这就是本节的视频点播（Video On Demand，VOD）。其实，视频点播程序与视频聊天程序有些类似，只不过在服务器提供的媒体数据方面，视频点播程序使用本地的媒体文件，而视频聊天程序通过实时采集获得。

本节将要实现一个点对点视频点播的演示程序VODDemo，它包括服务器和客户机两个部分。它们的程序界面如图3.21、图3.22所示。

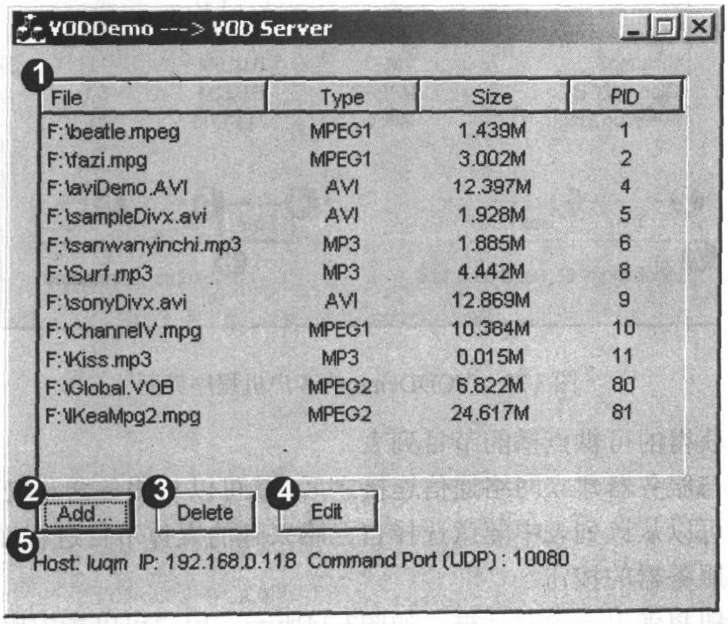


图 3.21 VODDemo 的服务器程序界面

① 服务器上提供给客户机点播的节目列表

服务器收集和管理本地机器上的媒体文件，并维持一个节目列表供客户机点播。节目列表包含的信息有：媒体文件名、媒体文件类型（MPEG1、MPEG2、MP3、AVI等）、媒体文件的大小、该媒体节目的惟一数字代号（Program ID，PID）。

② 增加一个媒体文件到节目列表中的按钮

单击本按钮将弹出一个标准的文件浏览对话框。用户可以浏览本地所有的媒体文件，并且选择一个加入到节目列表中去。确定选择某个媒体文件之后会弹出一个如图3.23所示的对话框，用户可以继续对该节目项进行编辑，如给它指定一个PID等。

③ 从节目列表中删除指定的媒体文件的按钮

用户首先要在节目列表中选中想要删除的一项，然后单击本按钮即可。

④ 编辑当前选择的节目项的按钮

用户首先要在节目列表中选中想要编辑的一项，然后单击本按钮弹出一个对话框，如图3.23所示，随后就可以进行编辑了，如改变当前节目项的PID等。

⑤ 服务器程序的状态信息栏

显示服务器的主机名、IP地址、命令控制端口等信息。

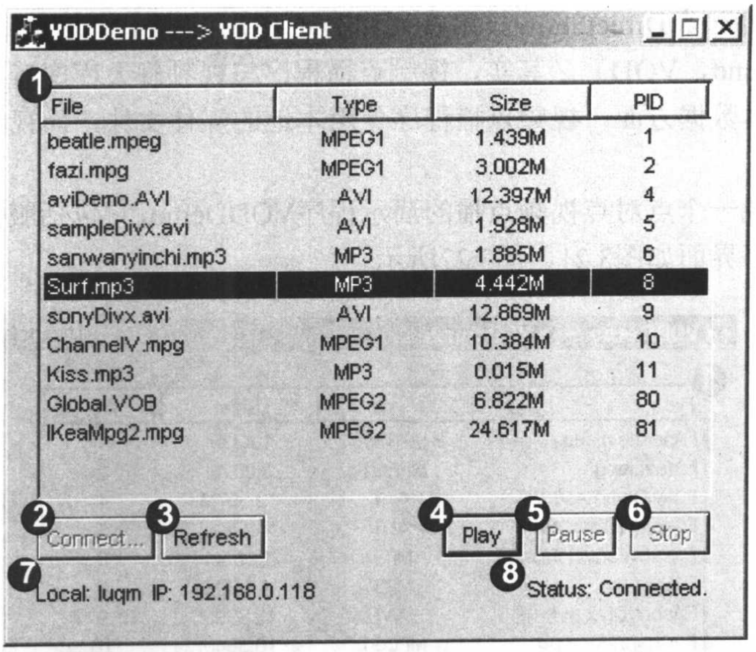


图 3.22　VODDemo 的客户机程序界面

① 从服务器获得的可供点播的节目列表

当客户机与服务器建立网络通信连接之后，就可以从服务器获取可供点播的节目列表。用户可以从该列表中随意选择自己感兴趣的媒体节目进行播放。

② 用于连接服务器的按钮

单击本按钮将弹出一个对话框，如图3.24所示。用户可以指定服务器的IP地址，然后尝试与服务器建立网络通信连接。

③ 从服务器获取最新的节目列表的按钮

服务器提供点播的节目列表可能变动，客户机提供本按钮，可以随时获取服务器上最新的节目列表。前提是，客户机与服务器已经成功建立了网络通信连接。

④ 指定的节目项的播放按钮

用户可以在客户机上显示的节目列表中随意选择一个自己感兴趣的媒体节目项，然后单击本按钮就可以开始播放了。

⑤ 当前节目项的暂停播放按钮

⑥ 当前节目项的停止播放按钮

⑦ 客户机的信息栏

用于显示客户机的主机名、IP地址等信息。

⑧ 客户机的状态栏

用于显示客户机程序的状态信息，如网络通信连接是否建立，媒体节目处于播放、暂停还是停止状态等。

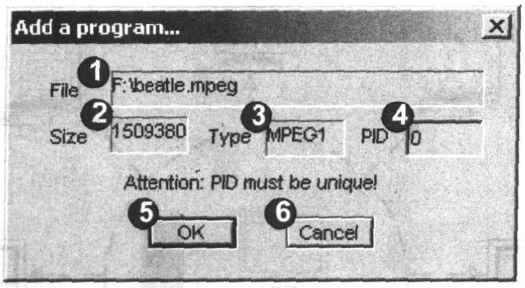


图 3.23 VODDemo 的服务器程序用于编辑节目项的界面

① 媒体文件的完整路径（用户指定） ② 媒体文件的大小（自动检测）
③ 媒体文件的类型（自动检测） ④ 该媒体节目的惟一数字代号（用户指定）
⑤ 确定将该媒体文件加入到节目列表中 ⑥ 放弃将该媒体文件加入到节目列表中

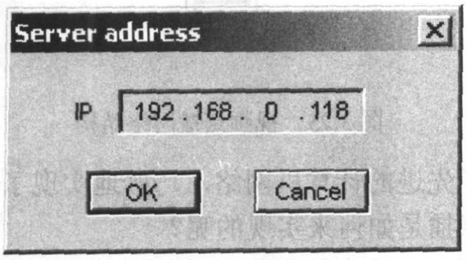


图 3.24 客户机连接时用于指定服务器 IP 地址的对话框

3.3.2 实现原理

所谓视频点播，就是根据用户自己的意愿，在一定量的可供选择的视频节目中间，想看什么节目就看什么节目。大家都有这样的体验：现行的广播电视是带有“强迫性”的，电视台按照它相对固定的节目列表来广播电视节目。用户在家里打开电视，对于某个电视台的节目只有选择看或不看的权利，一般无法要求电视台播出一个在它的节目列表之外的电视节目。视频点播恰恰满足了用户的这种需求，让用户真正“翻身”做了主人。

视频点播的实现首先需要搭建一个如图3.25所示的计算机网络。网络中配置一台高性能、大容量的视频服务器，专门用于收集、管理各种媒体文件。网络中的任一客户机都能与服务器建立连接，并且查看到服务器上的所有节目列表。用户根据自己的兴趣，通过客户机向服务器发送点播某个节目的指令；服务器在接收到该指令后，随即开始向这个客户机发送节目内容。就这样，服务器和客户机一个发送一个接收，并且客户机一边接收一边播放，从而真正满足了用户“点播”的要求。

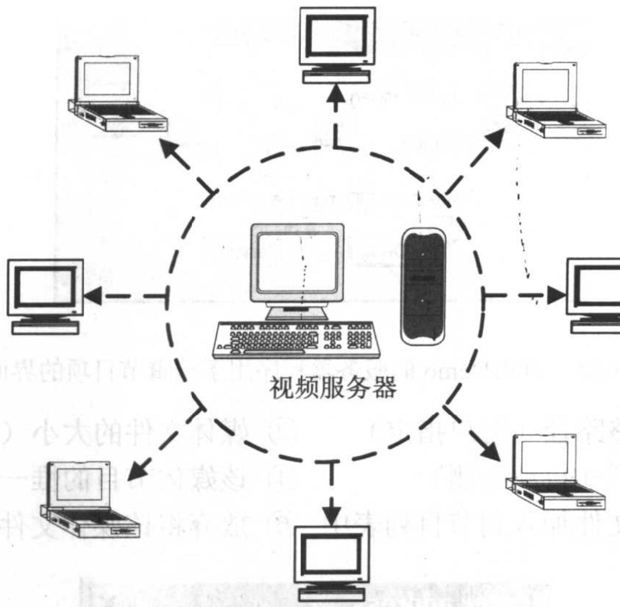


图 3.25 视频点播网络结构

视频点播实际上利用了先进的计算机网络，巧妙地实现了一种资源共享。那么，从程序开发的角度来看，视频点播是如何来实现的呢？

先来看服务器程序。服务器程序的实现似乎并不需要太多的多媒体技术，因为它更多地扮演一个媒体文件管理者的角色。服务器程序需要收集、管理本地的所有媒体文件，获取这些文件的描述信息，目的是维持一个节目列表供客户机查看和点播。另外，服务器程序还要使用网络通信技术，及时、准确地响应任一客户机的要求，并且按照要求控制特定媒体数据的发送。就这么简单！

然后来看客户机程序。如果单纯从与服务器建立连接、向服务器发送指令、接收来自服务器的媒体数据等方面考虑，似乎客户机程序的实现也不难（都是些网络通信方面的技术）。但客户机接收到的媒体数据如何实现同步播放呢？这就轮到DirectShow来大展拳脚了。大家知道，DirectShow中参与工作的基本形式是Filter。有一点是肯定的：客户机程序必须开发一个Source Filter用于接收网络数据，然后将这些网络数据引入到Filter Graph中去。问题是这个Source Filter怎么写？为了最大限度地使用DirectShow集成的一系列Filter，VODDemo演示程序将这个Source Filter写成了像标准的File Source Filter一样（被动地提供数据）。因为DirectShow提供的Splitter大都只能工作在拉模式下，这些Splitter的输入Pin上会专门使用一个线程到上一级Source Filter中去“拉”数据，而不支持Source Filter直接将数据“推”下来；将Source Filter写成拉模式后，播放网络接收来的媒体数据就与播放本地媒体文件一样了。图3.26、图3.27、图3.28、图3.29给出了VODDemo演示程序中，用于播放不同格式的网络媒体流的Filter Graph参考图。

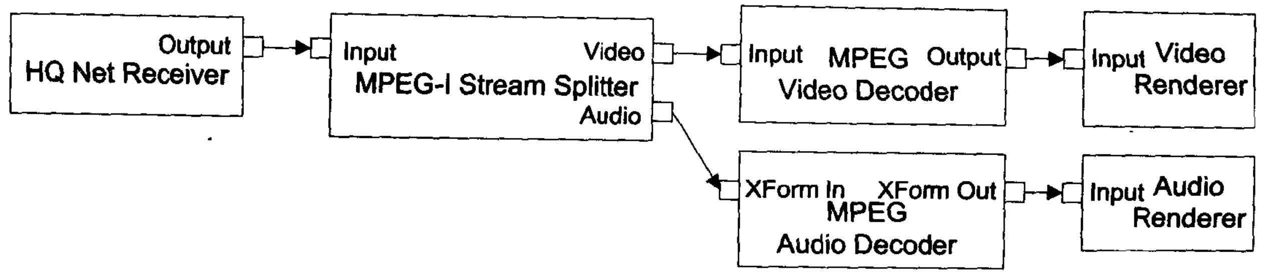


图 3.26 播放网络 MPEG1 数据的 Filter Graph

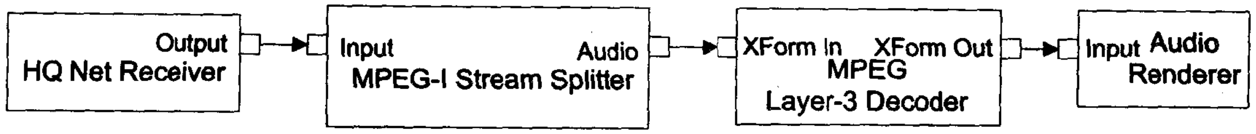


图 3.27 播放网络 MP3 数据的 Filter Graph

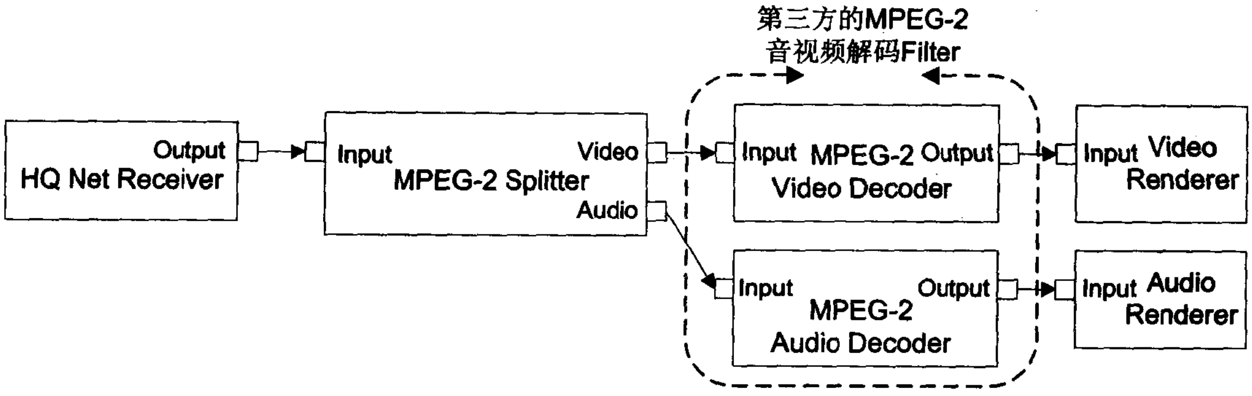


图 3.28 播放网络 MPEG2 数据的 Filter Graph

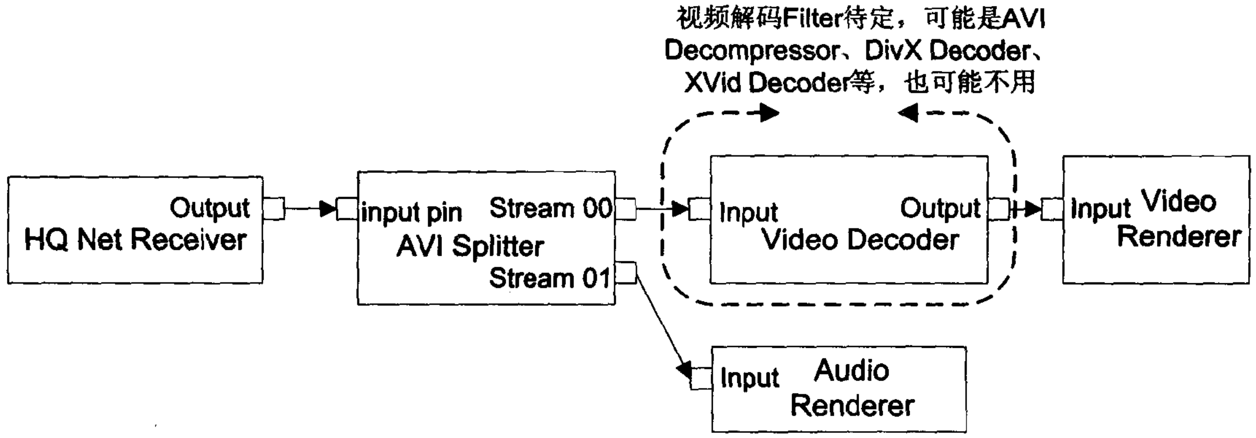


图 3.29 播放网络 AVI 数据的 Filter Graph

提示： 关于接收网络数据的Source Filter的开发，更多讨论参见本书附录A中的第40个问答“如何使用DirectShow播放实时流？”。

3.3.3 开发网络接收Filter

在3.2节的视频聊天程序中也实现了一个网络接收Filter。因为那个Filter接收的音视频数据是非压缩的，而且音视频是分离的，所以Source Filter在接收到数据后可以直接“推”给下一级Filter。本节将要开发的网络接收Filter可不能这么做！它必须工作在拉模式下，它提供数据的方式是被动的，而真正的数据驱动由下一级的Splitter来完成。

大家知道，拉模式的Source Filter在输出Pin上必须实现一个IAsyncReader接口，以便Source Filter内的数据被同步或异步读出。那么，怎么来实现这个接口呢？幸运的是，DirectShow SDK提供了一个实现拉模式Source Filter的例子：Async，它在SDK安装目录的Samples\C++\DirectShow\Filters\Async下。Async例子定义了一系列类，如用CAsyncStream类来处理数据来源问题，即当调用者要求读取一定量的数据到指定的缓存时，CAsyncStream类（它知道内部数据从哪里来、是否已经准备好）负责将要求的数据传送出去；CAsyncRequest类表示一个数据异步读取的请求；CAsyncIo类真正控制数据的输入输出（包括同步和异步两种方式的实现），外部调用者读取数据时直接与这个类打交道。另外，Async例子还实现了拉模式Source Filter的框架：实现了一个支持IAsyncReader接口的输出Pin类CAsyncOutputPin（从CBasePin类派生），以及一个使用CAsyncOutputPin作为输出Pin的Filter类CAsyncReader（从CBaseFilter类派生）。

提示：更多关于IAsyncReader接口的知识，请读者同时参考本书附录A中的第32个问答“为什么设计IAsncReader接口？CPullPin类有什么用？”。

在实现我们自己的拉模式Source Filter（取名为HQ Net Receiver）时，完全可以使用Async例子已经实现的一些类（只须将AsyncIo.h、AsyncIo.cpp、AsyncRdr.h、AsyncRdr.cpp 4个源文件复制到当前开发项目中来即可）。于是，开发任务变得轻松多了。我们需要做的仅仅是：从CAsyncStream类派生一个子类CNetStreamOut，以定制数据的来源（我们的数据是通过Socket从网络接收过来的）；从CAsyncReader类派生一个新的Filter类CFilterNetReceiver，以指定使用CNetStreamOut类作为数据源。（CFilterNetReceiver类还完成了其他的一些功能实现，如支持一个自定义接口INetReceiver等。）图3.30给出了HQ Net Receiver使用的类继承结构。

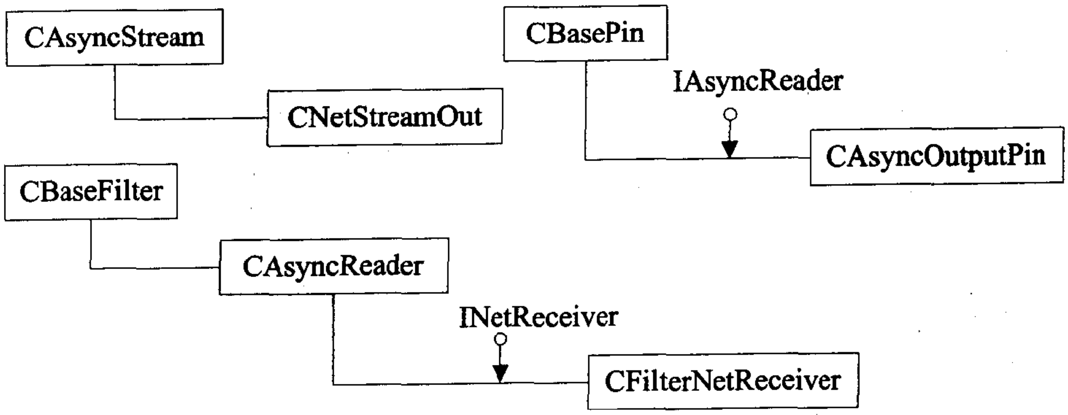


图 3.30 HQ Net Receiver 使用的类继承结构

下面就来介绍HQ Net Receiver的具体实现。先看CNetStreamOut类：这个类要实现父类CAsyncStream定义的所有纯虚函数。CNetStreamOut类使用了一个控制对象，将真正的与数据读取相关的操作都“委托”给了这个控制对象。因为控制对象负责接收网络数据，它当然更清楚当外部要求数据时如何将数据送出。

```
//
// CNetStreamOut.h
//

#ifndef __H_CNetStreamOut__
#define __H_CNetStreamOut__

#include "AsyncIo.h"
#include "CMediaController.h"

class CFilterNetReceiver;
class CNetStreamOut : public CAsyncStream
{
    friend class CFilterNetReceiver;
private:
    CCritSec            mStreamLock;    // 数据读取操作同步
    CMediaController    mController;    // 控制对象
public:
    CNetStreamOut(CFilterNetReceiver * inFilter);
    ~CNetStreamOut();
    // --- 实现CAsyncStream类所有的纯虚函数 ---
    // 设置在流中的偏移（准备读取新的数据）
    virtual HRESULT SetPointer(LONGLONG llPos);
    // 读取一定量的数据到指定的缓存中（外部调用此函数来获取数据）
    virtual HRESULT Read(PBYTE pbBuffer, DWORD dwBytesToRead,
        BOOL bAlign, LPDWORD pdwBytesRead);
    // 数据流总长度以及当前可以获得的数据长度
    virtual LONGLONG Size(LONGLONG *pSizeAvailable = NULL);
    // 数据对齐方式
    virtual DWORD Alignment();
    // 数据读取操作同步——锁定/解锁
    virtual void Lock();
    virtual void Unlock();
};

#endif // __H_CNetStreamOut__

//
// CNetStreamOut.cpp
//

#include <streams.h>
#include "CFilterNetReceiver.h"
#include "CNetStreamOut.h"

//////////////////////////////////////////////////////////////////////////
```

```
CNetStreamOut::CNetStreamOut(CFilterNetReceiver * inFilter) :
mController(inFilter)
{
}
CNetStreamOut::~CNetStreamOut()
{
}
// 设置在流中的偏移（委托给控制对象实现）
HRESULT CNetStreamOut::SetPointer(LONGLONG llPos)
{
    return mController.SetPointer(llPos);
}
// 读取一定量的数据到指定的缓存中（委托给控制对象实现）
HRESULT CNetStreamOut::Read(PBYTE pbBuffer, DWORD dwBytesToRead,
    BOOL bAlign, LPDWORD pdwBytesRead)
{
    return mController.Read(pbBuffer, dwBytesToRead, bAlign, pdwBytes
        Read);
}
// 数据流总长度以及当前可以获得的数据长度（委托给控制对象实现）
LONGLONG CNetStreamOut::Size(LONGLONG *pSizeAvailable)
{
    return mController.Size(pSizeAvailable);
}
// 数据按1字节对齐
DWORD CNetStreamOut::Alignment()
{
    return 1;
}
// 数据读取操作同步——锁定/解锁
void CNetStreamOut::Lock()
{
    mStreamLock.Lock();
}
void CNetStreamOut::Unlock()
{
    mStreamLock.Unlock();
}
```

再来看CFilterNetReceiver类的实现。因为拉模式下的数据读取框架已经在父类CAsyncReader中实现了；CFilterNetReceiver类需要做的仅仅是指定使用CNetStreamOut类来处理数据源问题。另外，Filter上还要实现一个自定义接口INetReceiver，以便于应用程序设置进来用于接收数据的Socket句柄，以及进行一些与数据接收相关的控制。

```
//
// INetReceiver.h
//

#ifndef __H_INetReceiver__
#define __H_INetReceiver__

#ifdef __cplusplus
```

```
extern "C" {
#endif

//--------------------------------------------------------------
// INetReceiver的各个接口函数定义
//--------------------------------------------------------------
DECLARE_INTERFACE_(INetReceiver, IUnknown)
{
    // 设置将要接收的网络数据类型
    STDMETHOD(SetStreamType) (THIS_
        long inType
    ) PURE;
    // 设置将要接收网络数据流的总长度，以及用于额外的格式校验的数据偏移量
    STDMETHOD(SetSize) (THIS_
        long inTotalSize, long inCheckOffset
    ) PURE;
    // 设置用于接收网络数据的Socket句柄
    STDMETHOD(SetStreamSocket) (THIS_
        SOCKET inSocket
    ) PURE;
    // 启动Socket接收线程开始接收数据
    STDMETHOD(StartReceiving) (THIS) PURE;
    // 停止Socket接收线程
    STDMETHOD(StopReceiving) (THIS) PURE;
    // 判断是否已经预接收了足够的数据，可以将Source Filter往下连接其他Filter了
    // 因为Source Filter与Spiltter连接时，Splitter会要求一定的数据进行格式校验
    STDMETHOD(CanConnectToDownstream) (THIS) PURE;
};
//--------------------------------------------------------------

#ifdef __cplusplus
}
#endif

#endif // __H_INetReceiver__

//
// CFilterNetReceiver.h
// Desc: Filter and interfaces implementation.
//

#ifndef __H_CFilterNetReceiver__
#define __H_CFilterNetReceiver__

#include "AsyncIo.h"
#include "AsyncRdr.h"
#include "INetReceiver.h"
#include "CNetStreamOut.h"

class CFilterNetReceiver : public CAsyncReader, public INetReceiver
{
private:
```

```
    CNetStreamOut    mNetStream;  // 数据源处理对象
private:
    CFilterNetReceiver(LPUNKNOWN lpunk, HRESULT *phr);
    ~CFilterNetReceiver();
public:
    static CUnknown * WINAPI CreateInstance(LPUNKNOWN lpunk, HRESULT *phr);
    DECLARE_IUNKNOWN;
    // 暴露本Filter实现的自定义接口
    STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void ** ppv);
    // 重写Filter上的运行、暂停、停止函数
    STDMETHODIMP Run(REFERENCE_TIME tStart);
    STDMETHODIMP Pause();
    STDMETHODIMP Stop();
    // 当Filter开始运行时调用的函数（可以进行一些初始化）
    HRESULT StartStreaming(void);
    // 当Filter停止运行时调用的函数（可以进行一些初始化）
    HRESULT StopStreaming(void);
    // 根据接收的网络数据类型决定输出Pin上使用的媒体类型
    HRESULT DecideOutputMediaType(long inMedia);
    BOOL IsFlushing(void);
    // 实现对Filter Graph的控制
    void PauseGraph(void);
    void RunGraph(void);

    // --- INetReceiver的各个接口函数声明 ---
    STDMETHODIMP SetStreamType(long inType);
    STDMETHODIMP SetSize(long inTotalSize, long inCheckOffset);
    STDMETHODIMP SetStreamSocket(SOCKET inSocket);
    STDMETHODIMP StartReceiving(void);
    STDMETHODIMP StopReceiving(void);
    STDMETHODIMP CanConnectToDownstream(void);
};

#endif // __H_CFilterNetReceiver__

//
// CFilterNetReceiver.cpp
//

#include <streams.h>
#pragma warning(disable: 4511 4512)
#pragma warning(disable: 4355)

#include <initguid.h>
#include "FilterNetReceiverGuid.h"

#if (1100 > _MSC_VER)
#include <olectlid.h>
#else
#include <olectl.h>
#endif
```

```
#include "CFilterNetReceiver.h"
#include "defines.h"

//
// 填写Filter的注册信息
//
const AMOVIESETUP_MEDIATYPE sudPinTypes =
{
   &MEDIATYPE_Video,            // Major type
   &MEDIASUBTYPE_NULL           // Subtype
};

const AMOVIESETUP_PIN psudPins[] =
{
  {
        L"Output",          // String pin name
        FALSE,              // Is it rendered
        TRUE,               // Is it an output
        FALSE,              // Allowed none
        FALSE,              // Allowed many
        &CLSID_NULL,        // Connects to filter
        L"Input",           // Connects to pin
        1,                  // Number of types
        &sudPinTypes        // The pin details
   }
};

const AMOVIESETUP_FILTER sudFilter =
{
   &CLSID_NetReceiver,       // Filter CLSID
   L"HQ Net Receiver",       // Filter name
   MERIT_DO_NOT_USE,         // Its merit
   1,                        // Number of pins
   psudPins                  // Pin details
};

CFactoryTemplate g_Templates[] =
{
   {
        L"HQ Net Receiver",
        &CLSID_NetReceiver,
        CFilterNetReceiver::CreateInstance,
        NULL,
        &sudFilter
    }
};
int g_cTemplates = sizeof(g_Templates) / sizeof(g_Templates[0]);

// ---------------------------------------------------------------------
//          Filter implementation
// ---------------------------------------------------------------------
CFilterNetReceiver::CFilterNetReceiver(LPUNKNOWN lpunk, HRESULT *phr) :
```

```
CAsyncReader(NAME("Net_Receiver"), lpunk, &mNetStream, phr),
mNetStream(this)
{
    m_clsid = CLSID_NetReceiver;
    // 假设接收的网络数据是MPEG1格式的
    DecideOutputMediaType(FT_MPEG1);
}
CFilterNetReceiver::~CFilterNetReceiver()
{
}
// Filter组件对象的创建函数
CUnknown * WINAPI CFilterNetReceiver::CreateInstance(LPUNKNOWN punk,
    HRESULT *phr)
{
    CFilterNetReceiver *pNewObject = new CFilterNetReceiver(punk, phr);
    if (pNewObject == NULL)
    {
        *phr = E_OUTOFMEMORY;
    }
    return pNewObject;
}
// 暴露本Filter实现的自定义接口
STDMETHODIMP CFilterNetReceiver::NonDelegatingQueryInterface(REFIID riid,
    void ** ppv)
{
    CheckPointer(ppv, E_POINTER);
    if (riid == IID_INetReceiver)
    {
        return GetInterface((INetReceiver *) this, ppv);
    }
    else
    {
        return CAsyncReader::NonDelegatingQueryInterface(riid, ppv);
    }
}
// Filter进入运行状态时调用如下Run函数
STDMETHODIMP CFilterNetReceiver::Run(REFERENCE_TIME tStart)
{
    CAutoLock lck(&m_csFilter);   // Filter对象上的同步
    return CAsyncReader::Run(tStart);
}
// Filter进入暂停状态时调用如下Pause函数
STDMETHODIMP CFilterNetReceiver::Pause()
{
    CAutoLock lck(&m_csFilter);   // Filter对象上的同步
    HRESULT hr = NOERROR;
    if (m_State == State_Paused)
    {
        // 如果现在已经是暂停状态了，不用做任何操作
    }
    else
    {
```

```
        if (m_State == State_Stopped)
        {
            // 如果是从停止状态转入暂停状态，则调用StartStreaming函数进行一些
            // 初始化
            hr = StartStreaming();
        }
        else if (m_State == State_Running)
        {
            // 如果是从运行状态转入暂停状态……
        }

        if (SUCCEEDED(hr))
        {
            hr = CBaseFilter::Pause();
        }
    }
    return hr;
}
// Filter进入停止状态时调用如下Stop函数
STDMETHODIMP CFilterNetReceiver::Stop()
{
    CAutoLock lck1(&m_csFilter);  // Filter对象上的同步
    HRESULT hr = CAsyncReader::Stop();
    if (SUCCEEDED(hr))
    {
        // 调用StopStreaming函数进行一些初始化
        hr = StopStreaming();
    }
    return hr;
}
HRESULT CFilterNetReceiver::StartStreaming(void)
{
    // 告诉控制对象：Filter Graph开始运行了
    mNetStream.mController.StartStreaming();
    return NOERROR;
}
HRESULT CFilterNetReceiver::StopStreaming(void)
{
    // 告诉控制对象：Filter Graph停止运行了
    mNetStream.mController.StopStreaming();
    return NOERROR;
}
// 根据接收的网络数据类型决定输出Pin上使用的媒体类型
HRESULT CFilterNetReceiver::DecideOutputMediaType(long inMedia)
{
    // 主媒体类型总是MEDIATYPE_Stream
    m_mt.SetType(&MEDIATYPE_Stream);
    switch (inMedia)
    {
    case FT_MPEG1:   // MPEG1数据
        m_mt.SetSubtype(&MEDIASUBTYPE_MPEG1System);
        break;
```

```
    case FT_MPEG2:  // MPEG2数据
        m_mt.SetSubtype(&MEDIASUBTYPE_MPEG2_PROGRAM);
        break;
    case FT_MP3:    // MPEG3数据
        m_mt.SetSubtype(&MEDIASUBTYPE_MPEG1Audio);
        break;
    case FT_AVI:    // AVI数据
        m_mt.SetSubtype(&MEDIASUBTYPE_Avi);
        break;
    default:
        m_mt.SetSubtype(&MEDIASUBTYPE_NULL);
        return E_FAIL;
    }
    return NOERROR;
}
// 判断是不是正在“清场”
BOOL CFilterNetReceiver::IsFlushing(void)
{
    return m_Io.m_bFlushing;
}
// 暂停Filter Graph的运行
void CFilterNetReceiver::PauseGraph(void)
{
    if (m_pGraph)
    {
        if (m_State == State_Paused)
        {
            return;
        }
        // 获得Filter Graph Manager上的IMediaControl接口
        IMediaControl * pControl = NULL;
        m_pGraph->QueryInterface(IID_IMediaControl, (void**) &pControl);
        if (pControl)
        {
            pControl->Release();
            pControl->Pause();
        }
    }
}
// 运行Filter Graph
void CFilterNetReceiver::RunGraph(void)
{
    if (m_pGraph)
    {
        if (m_State == State_Running)
        {
            return;
        }
        // 获得Filter Graph Manager上的IMediaControl接口
        IMediaControl * pControl = NULL;
        m_pGraph->QueryInterface(IID_IMediaControl, (void**) &pControl);
        if (pControl)
```

```
            {
                pControl->Release();
                pControl->Run();
            }
        }
}
// --- INetReceiver的各个接口函数实现 ---
// 设置将要接收的网络数据类型，然后决定输出Pin上相应使用的媒体类型
STDMETHODIMP CFilterNetReceiver::SetStreamType(long inType)
{
    mNetStream.mController.SetStreamType(inType);
    // 根据网络数据类型决定输出Pin上使用的媒体类型
    DecideOutputMediaType(inType);
    return NOERROR;
}
// 给控制对象设置将要接收网络数据流的总长度，以及用于额外的格式校验的数据偏移量
STDMETHODIMP CFilterNetReceiver::SetSize(long inTotalSize, long
    inCheckOffset)
{
    mNetStream.mController.SetSize(inTotalSize, inCheckOffset);
    return NOERROR;
}
// 给控制对象设置用于接收网络数据的Socket句柄
STDMETHODIMP CFilterNetReceiver::SetStreamSocket(SOCKET inSocket)
{
    mNetStream.mController.SetStreamSocket(inSocket);
    return NOERROR;
}
// 启动控制对象中的网络数据接收线程
STDMETHODIMP CFilterNetReceiver::StartReceiving(void)
{
    BOOL pass = mNetStream.mController.StartReceiving();
    return pass ? NOERROR : E_FAIL;
}
// 停止控制对象中的网络数据接收线程
STDMETHODIMP CFilterNetReceiver::StopReceiving(void)
{
    mNetStream.mController.StopReceiving();
    return NOERROR;
}
// 判断是否已经预接收了足够的数据，可以将Source Filter往下连接其他Filter了
STDMETHODIMP CFilterNetReceiver::CanConnectToDownstream(void)
{
    BOOL ready = mNetStream.mController.IsReadToBeChecked();
    return ready ? S_OK : S_FALSE;
}

/******************************Public Routine******************************\
* DLL导出函数：用于Filter的注册和反注册
\**************************************************************************/
STDAPI DllRegisterServer()
{
```

```
    return AMovieDllRegisterServer2( TRUE );
}
STDAPI DllUnregisterServer()
{
    return AMovieDllRegisterServer2( FALSE );
}

#pragma warning( disable:4514)
```

其实，Filter框架的实现很简单，而且也应该这样简单，因为Filter毕竟只是个“框架”。关键的实现还要看控制类CMediaController：

```
//
// CMediaController.h
//

#ifndef __H_CMediaController__
#define __H_CMediaController__

#include "CStreamSocket2.h"

class CFilterNetReceiver;
class CMediaController
{
private:
    CFilterNetReceiver  * mReceiveFilter; // Filter对象指针
    CCritSec     mSyncBuffer;        // 数据缓存读写操作的同步
    BYTE *       mMediaBuffer;       // 数据缓存指针
    long         mBufferSize;        // 数据缓存大小
    long         mMinWorkingSize;    // 缓存中的有效数据允许的最小值
                                     // 当缓存中的有效数据小于这个值时进行缓存
                                     // 整理
    long         mReadingOffset;     // 从缓存中读数据的起始地址
    long         mWritingOffset;     // 向缓存中写数据的起始地址
    long         mCheckingSize;      // 预接收数据的大小（用于Filter连接时格式
                                     // 校验）
    long         mAvailabeSize;      // 当前可以获取的数据大小
    long         mTotalSize;         // 数据流总长度
    BOOL         mIsWaitingToRead;   // 是否正在等待读取足够的数据当缓存中没有足
                                     // 够的数据时，则让外部调用者等待
    BOOL         mIsWaitingToAddData;    // 每次接收到的网络数据都要加到缓存中
                                         //去，如果此时缓存没有足够的空间则等待
    long         mStreamType;            // 网络数据类型
    BYTE *       mExtraBuffer;           // 用于格式校验的额外数据缓存
    long         mExtraCheckOffset;      // 额外格式数据在整个数据流中的偏移
                                         // 位置
    long         mExtraBufferSize;       // 额外缓存的大小
    long         mExtraWritingOffset;    // 额外缓存中写数据的偏移地址
    long         mExtraReadingOffset;    // 额外缓存中读数据的偏移地址
    BOOL         mReadFromExtraBuffer;   // 指示当前是否在额外缓存中读数据
    BOOL         mIsReadyToBeChecked;    // 指示是否接收到了预定的数据量
    BOOL         mCheckingFinished;      // 指示Filter连接时的格式校验是否结束
```

```
    BOOL        mIsEOS;                        // 指示是否所有的网络数据都已经接收
                                               // 完毕
    CStreamSocket2 mStrmSocket;                // 用于接收网络数据的Socket包装对象

public:
    CMediaController(CFilterNetReceiver * inFilter);
    ~CMediaController();
    // 当Filter Graph开始运行、停止时进行一些初始化
    void StartStreaming(void);
    void StopStreaming(void);
    // Filter上的INetReceiver接口“委托”过来的函数
    void SetStreamType(long inType);
    void SetStreamSocket(SOCKET inSocket);
    void SetSize(long inTotalSize, long inCheckOffset);
    BOOL StartReceiving(void);
    void StopReceiving(void);
    // 状态检查：是否已经接收到了预定的数据量?
    // 如果是，可以让上层应用程序完成Filter的连接了
    BOOL IsReadToBeChecked(void);
    // 将接收到的网络数据加入到缓存中去
    BOOL AddData(char * inData, long inLength);
    // 当所有的网络数据接收完毕后调用EndOfStream函数告诉控制对象
    void EndOfStream(void);
    // 数据源对象（CNetStreamOut类的实例）“委托”过来的函数
    HRESULT SetPointer(LONGLONG llPos);
    HRESULT Read(PBYTE pbBuffer, DWORD dwBytesToRead,
        BOOL bAlign, LPDWORD pdwBytesRead);
    LONGLONG Size(LONGLONG *pSizeAvailable);
};

#endif // __H_CMediaController__
```

下面的章节将结合支持不同的网络数据类型，分别阐述CMediaController类的具体实现。

提示：本书配套光盘的SourceCodes\Chapter03\VODDemo\FilterNetReceiver目录下提供了HQ Net Receiver Filter的完整实现。

3.3.4 支持MPEG1、MP3

DirectShow集成的一系列Filter中，提供了对MPEG1数据解码的全套Filter（包括MPEG-1 Stream Splitter、MPEG Video Decoder、MPEG Audio Decoder等）。因此播放MPEG1数据，至少从系统中的Filter组件配备方面，构建如图3.26所示的Filter Graph是没有问题的。

为了保证Source Filter能够与MPEG-1 Stream Splitter相连，Source Filter的输出Pin上应该使用媒体类型（MEDIATYPE_Stream，MEDIASUBTYPE_MPEG1System）。不过，仅仅这么做还是不够的。因为Splitter在与Source Filter连接的时候，会要求从Source Filter中读取一部分数据进行格式校验（获取音视频属性，如图像的宽高、帧率、音频的采样频率等）。一来验证数据流的类型，二来也方便Splitter后续Filter的连接。这就要求Source Filter在与

Splitter开始连接之前，预先接收一定量的数据进行缓存。

控制类CMediaController使用CStreamSocket2类的一个实例来负责网络数据的接收。这个CStreamSocket2类似曾相识，但实际上它与3.1.1节中的CStreamSocket类有些不一样，主要的区别在于增加了一个接口函数设置使用它的控制对象指针，还有就是接收过程的具体化（与控制对象上的操作关联）。CStreamSocket2类中的网络数据接收过程如下：

```
//
// CStreamSocket2.cpp
//
// 接收网络数据的循环过程
void CStreamSocket2::ReceivingLoop(void)
{
    char   buf[PACK_SIZE];   // 接收缓存
    int    toread = PACK_SIZE;
    int    bytes  = 0;

    while (mIsReceiving)
    {
        bytes = recv(mSocket, buf, toread, 0);
        if (bytes == 0)
        {
        //  TRACE("socket closed\n");
            mIsReceiving = FALSE;
            break;
        }
        else if (bytes == SOCKET_ERROR)
        {
        //  TRACE("recv socket error, code:%d\n", WSAGetLastError());
            mIsReceiving = FALSE;
            break;
        }
        // 将接收到的网络数据送给控制对象保存
        mIsReceiving = mController->AddData(buf, bytes);
    }
    // 通知控制对象：所有的网络数据接收完毕
    mController->EndOfStream();
}
```

下面来看CMediaController类的具体实现。

```
//
// CMediaController.cpp
//

#include <streams.h>
#include "CFilterNetReceiver.h"
#include "CMediaController.h"
#include "defines.h"

//////////////////////////////////////////////////////////////////////////////
CMediaController::CMediaController(CFilterNetReceiver * inFilter)
```

```
{
    mReceiveFilter       = inFilter;
    ASSERT(mReceiveFilter);
    // 申请10M空间用于缓存网络数据（mBufferSize的值可根据实际情况调整）
    mBufferSize          = 10 * 1024 * 1024;
    // 缓存中的有效数据允许的最小值（暂定为20K），当缓存中的有效数据小于这个值时将进
    // 行缓存整理，即将有效数据从缓存的尾部搬移到缓存的头部，以留出空间继续接收数据
    mMinWorkingSize      = 20 * 1024;
    // 用于Filter连接而预接收的数据大小（暂定为30K）
    // （mCheckingSize的值可根据实际情况调整）
    mCheckingSize        = 30 * 1024;
    // 申请缓存
    mMediaBuffer         = new BYTE[mBufferSize];
    // 参数初始化
    mWritingOffset       = 0;  // 写数据指针指向缓存的开头
    mReadingOffset       = 0;  // 读数据指针指向缓存的开头
    mAvailabeSize        = 0;
    mIsReadyToBeChecked = FALSE;
    mCheckingFinished    = FALSE;
    mIsEOS               = FALSE;
    mStreamType          = 0;
    mTotalSize           = 0;
    mExtraCheckOffset    = 0;
    mExtraBuffer         = NULL;
    mExtraBufferSize     = 0;
    mExtraWritingOffset = 0;
    mExtraReadingOffset = 0;
    mReadFromExtraBuffer     = FALSE;
    mIsWaitingToRead         = FALSE;
    mIsWaitingToAddData      = FALSE;
    // 将本控制对象指针设置给Socket包装对象
    mStrmSocket.SetController(this);
}
CMediaController::~CMediaController()
{
    // 释放使用过的内存
    if (mMediaBuffer)
    {
        delete[] mMediaBuffer;
        mMediaBuffer = NULL;
    }
    if (mExtraBuffer)
    {
        delete[] mExtraBuffer;
        mExtraBuffer = NULL;
    }
}
// 当Filter Graph开始运行时进行一些初始化
void CMediaController::StartStreaming(void)
{
    mIsEOS                  = FALSE;
    mCheckingFinished       = TRUE;
```

```
    mReadFromExtraBuffer     = FALSE;
    mIsWaitingToRead         = FALSE;
    mIsWaitingToAddData      = FALSE;
    CAutoLock   lck(&mSyncBuffer);
    mReadingOffset           = 0;
    mExtraReadingOffset      = 0;
}
// 当Filter Graph停止时进行一些初始化
void CMediaController::StopStreaming(void)
{
    mAvailabeSize   = 0;
    CAutoLock   lck(&mSyncBuffer);
    mWritingOffset  = 0;
    mReadingOffset  = 0;
}
// 设置网络数据的类型
void CMediaController::SetStreamType(long inType)
{
    mStreamType = inType;
}
// 设置用于接收网络数据的Socket句柄
void CMediaController::SetStreamSocket(SOCKET inSocket)
{
    // 将Socket句柄关联到包装对象中
    mStrmSocket.Attach(inSocket);
}
// 设置网络数据流的总长度，以及额外格式数据的偏移量
void CMediaController::SetSize(long inTotalSize, long inCheckOffset)
{
    mTotalSize          = inTotalSize;
    mExtraCheckOffset   = inCheckOffset;
    // 根据数据类型的不同，有些需要额外的格式信息，有些不需要，如果mExtraCheckOffset
    // 偏移量大于0，说明需要额外的格式信息，则需为接收这些格式数据分配额外缓存；否则就
    // 不需要
    if (mExtraCheckOffset > 0)
    {
        // 代码省略……
    }
    // 格式校验需要的数据量不可能超过数据流的总长度
    if (mCheckingSize > mTotalSize)
    {
        mCheckingSize = mTotalSize;
    }
}
// 启动Socket数据接收线程
BOOL CMediaController::StartReceiving(void)
{
    return mStrmSocket.StartReceiving();
}
// 停止Socket数据接收线程
void CMediaController::StopReceiving(void)
{
```

```
    mStrmSocket.StopReceiving();
}
// 状态检查：是否已经接收到了预定的数据量
BOOL CMediaController::IsReadToBeChecked(void)
{
    return mIsReadyToBeChecked;
}
// 当从网络上接收到一块数据后，调用该函数将数据加入到控制对象内部的缓存中
BOOL CMediaController::AddData(char * inData, long inLength)
{
    // 如果一种数据类型需要额外的格式信息才能连接完成Filter的连接，则服务器会首先发送
    // 这些额外的格式数据。接收时，也应该首先将这些额外的格式数据放到专门用于保存额外
    // 格式数据的缓存中
    if (mReceiveFilter->IsStopped() && mExtraCheckOffset > 0 &&
        mExtraWritingOffset < mExtraBufferSize)
    {
        // 代码省略……
    }

    // 检查当前缓存是否有足够的空间来容纳进来的数据，如果没有足够的空间则等待（等待
    // Splitter的数据读出线程读出更多的数据）
    while (mBufferSize - mWritingOffset < inLength)
    {
        mIsWaitingToAddData = TRUE;
        if (mReceiveFilter->IsFlushing())
        {
            mIsWaitingToAddData = FALSE;
            return FALSE;
        }

        Sleep(2);
        {
            // --- 整理缓存 ---
            // 当缓存中的有效数据不多时，将有效数据挪到缓存的头部，这样就可以留出空间
            // 来接收新进来的网络数据
            CAutoLock  lck(&mSyncBuffer);
            long validLength = mWritingOffset - mReadingOffset;
            if (mCheckingFinished)
            {
                if (validLength <= mMinWorkingSize || mIsWaitingToRead)
                {
                    // 有效数据在缓存中的搬移
                    memmove(mMediaBuffer, mMediaBuffer+mReadingOffset,
                        validLength);
                    mReadingOffset = 0;
                    mWritingOffset = validLength;
                    mAvailabeSize  = validLength;
                }
            }
        }
    }
    mIsWaitingToAddData = FALSE;
```

```
    CAutoLock  lck(&mSyncBuffer);
    // 将新进来的网络数据加到缓存中保存
    memcpy(mMediaBuffer + mWritingOffset, inData, inLength);
    mWritingOffset += inLength;  // 更新写数据偏移量
    mAvailabeSize  += inLength;

    // 如果当前接收到的数据量已经达到了Filter连接需要的预接收量，
    // 则设置标记量mIsReadyToBeChecked为有效
    if (!mCheckingFinished && !mIsReadyToBeChecked && mWritingOffset >=
        mCheckingSize)
    {
        mIsReadyToBeChecked = TRUE;
    }
    return TRUE;
}
// 网络数据都已经接收完毕的通知
void CMediaController::EndOfStream(void)
{
    mIsEOS = TRUE;
}
// 设置数据流中的偏移量，准备下一次数据的（随机）读取
HRESULT CMediaController::SetPointer(LONGLONG llPos)
{
    // 仅在Filter连接时的格式校验过程支持SetPointer功能，正常数据流播放过程中的偏移
    // 计算由Read函数完成
    if (!mCheckingFinished && mReceiveFilter->IsStopped())
    {
        if (llPos >= 0 && llPos < mBufferSize)
        {
            CAutoLock  lck(&mSyncBuffer);
            mReadingOffset = long(llPos);
            mReadFromExtraBuffer = FALSE;
        }
    }
    // 其他代码省略
    // ……
    return NOERROR;
}
// 外部调用者想要读取dwBytesToRead大小的数据到pbBuffer缓存中，Read函数返回时指定
// pdwBytesRead为实际读到的数据量
HRESULT CMediaController::Read(PBYTE pbBuffer, DWORD dwBytesToRead,
    BOOL bAlign, LPDWORD pdwBytesRead)
{
    // 如果要求读取额外的格式数据，则从保存额外格式数据的缓存中读取
    if (mExtraCheckOffset > 0 && mReceiveFilter->IsStopped() &&
        mReadFromExtraBuffer)
    {
        // 代码省略……
        return NOERROR;
    }
```

```
        // 如果没有足够的数据供读出，则等待接收更多的网络数据
        while (mWritingOffset - mReadingOffset < dwBytesToRead)
        {
            mIsWaitingToRead = TRUE;
            if (mReceiveFilter->IsFlushing() || mIsEOS)
            {
                mIsWaitingToRead = FALSE;
                *pdwBytesRead = 0;
                return NOERROR;
            }
            Sleep(2);
        }
        mIsWaitingToRead = FALSE;
        // 从缓存中读出要求大小的数据
        CAutoLock  lck(&mSyncBuffer);
        memcpy(pbBuffer, mMediaBuffer + mReadingOffset, dwBytesToRead);
        mReadingOffset += dwBytesToRead;  // 更新读数据偏移量
        *pdwBytesRead   = dwBytesToRead;
        return NOERROR;
    }
    // 得到数据流的总长度以及当前可以获得的数据长度
    LONGLONG CMediaController::Size(LONGLONG *pSizeAvailable)
    {
        if (pSizeAvailable)
        {
            *pSizeAvailable = mAvailabeSize;
        }
        return mTotalSize;
    }
```

因为MPEG-1 Stream Splitter在与Source Filter连接的时候要求的数据，只需数据流开头的那部分就可以了。因此在上述CMediaController类的实现中，凡是牵涉到额外格式数据的代码都暂时省略了（这部分代码将在支持MPEG2的时候介绍）。关于CMediaController类中使用的数据缓存，这里还要解释一下。这是一个较大容量的连续内存区域，写操作的地址通过mWritingOffset来控制，读操作的地址通过mReadingOffset来控制；缓存的读写操作通过对mSyncBuffer的加锁/解锁来同步。从mReadingOffset到mWritingOffset这段区域的数据称为有效数据；随着网络数据的接收以及缓存中的数据不断被Splitter读出，mReadingOffset和mWritingOffset指向缓存中的位置都向缓存的尾部靠近。当mWritingOffset十分靠近缓存的尾部，以致于没有足够的空间容纳新进来的网络数据时，网络数据接收线程将会阻塞，以等待数据读取线程继续读出数据，直到mReadingOffset也向缓存的尾部足够靠近，mReadingOffset到mWritingOffset的有效数据长度小于最小值mMinWorkingSize时，可以整理缓存，将有效数据从缓存的尾部搬移到缓存的头部，同时更新mReadingOffset和mWritingOffset的值（也移到缓存的头部去了）。于是，新一轮的数据接收、读出又可以开始了。

当然，上述的缓存数据接收、读出过程中还会出现一种情况，就是如果网络通信状况不好，导致缓存中的有效数据量跟不上Splitter的读取需求，则Splitter的数据读取线程也会被阻塞，以等待接收足够的网络数据。这种情况下，Filter Graph仍然在运行，Spiltter经过

等待一段时间后尽管得到了想要的数据，但这部分数据经过解码后再送到Renderer Filter的时候可能已经“过时”了。于是，Video Renderer会发出一个质量控制（Quality Control）消息，并且这个消息最终由Splitter来处理。此时Splitter可能会要求对Source Filter内的数据流进行重新定位。如果出现这种情况就麻烦了，因为HQ Net Receiver并没有处理这方面的问题，有兴趣的读者可以继续深入研究一下。

接下去来看一下支持MP3的情况。MP3是一种仅含有第三层（Layer-3）MPEG1音频编码的数据类型，可以算是MPEG1数据的一种特例。因此支持MP3的播放，除了要将Source Filter的输出Pin上使用的媒体类型改为（MEDIATYPE_Stream，MEDIASUBTYPE_MPEG1 Audio）之外，只需构建如图3.27所示的Filter Graph就可以了，其他的与上述处理普通MPEG1数据的方式一样，这里不再赘述。

3.3.5 支持MPEG2

DirectShow集成的一系列Filter中，与MPEG2相关的有两个：MPEG2 Splitter和MPEG2 Demulitplexer。这两个Filter都是用于将MPEG2流中的音视频分离的。那么，MPEG2的音视频解码Filter呢？DirectShow没有集成！因此，如果想在客户机上播放MPEG2类型的媒体节目，不预先安装第三方的MPEG2音视频解码Filter是不行的，否则如图3.28所示的Filter Graph将无法成功构建。

了解了上述前提条件之后，我们再来看程序中对支持MPEG2作出的处理。为了保证Source Filter能够与MPEG2 Splitter相连，Source Filter的输出Pin上应该使用媒体类型（MEDIATYPE_Stream，MEDIASUBTYPE_MPEG2_PROGRAM）。不过，仅仅这么做还是不够的。通过研究发现，MPEG2 Splitter在与Source Filter连接的时候，读取的格式校验数据量比较大，而且除了数据流头部的数据外，还要求读取数据流尾部1M大小的数据（上文中称为额外格式数据）。这怎么办呢？正常情况下，服务器是将指定媒体文件的内容从头到尾顺序发送过来的，难道要客户机等到数据流全部接收完之后再开始Filter的连接和播放？用户有耐心等这么久吗？

解决这个问题的方法其实也很简单，可以让服务器先把媒体文件尾部的额外格式数据发送给客户机！CMediaController类为支持MPEG2而做出的处理如下：

```
//
// CMediaController.cpp
//
// 设置网络数据流的总长度，以及额外格式数据的偏移量
void CMediaController::SetSize(long inTotalSize, long inCheckOffset)
{
    mTotalSize        = inTotalSize;
    mExtraCheckOffset = inCheckOffset;
    // 如果偏移量大于0，说明Filter连接时需要额外的格式信息，则需为接收这些格式数据分
    // 配一个缓存
    if (mExtraCheckOffset > 0)
    {
        // 通过额外格式数据在整个数据流中的偏移量，计算出额外格式数据的大小
        mExtraBufferSize  = inTotalSize - inCheckOffset;
```

```
        // 为保存额外格式数据而申请一块内存区域
        if (mExtraBuffer)
        {
            delete[] mExtraBuffer;
            mExtraBuffer = NULL;
        }
        mExtraBuffer = new BYTE[mExtraBufferSize];
    }
    // 调整一些参数，如网络数据类型是MPEG2时，将预接收用于Filter连接时格式校验的数
    // 据量调整到3M
    if (mStreamType == FT_MPEG2)
    {
        mCheckingSize = 3 * 1024 * 1024;
    }
    if (mCheckingSize > mTotalSize)
    {
        mCheckingSize = mTotalSize;
    }
}
// 当从网络上接收到一块数据后，调用该函数将数据加入到控制对象内部的缓存中
BOOL CMediaController::AddData(char * inData, long inLength)
{
    // 如果一种数据类型需要额外的格式信息才能连接完成Filter的连接，则服务器会首先发送
    // 这些额外的格式数据。这里接收时，也应该首先将这些额外的格式数据放到专门用于保存
    // 额外格式数据的缓存中
    // mExtraCheckOffset > 0表示这种数据类型需要额外的格式数据支持
    // mExtraWritingOffset < mExtraBufferSize表示额外的格式数据还没有接收完
    if (mReceiveFilter->IsStopped() && mExtraCheckOffset > 0 &&
        mExtraWritingOffset < mExtraBufferSize)
    {
        long space = mExtraBufferSize - mExtraWritingOffset;
        if (space >= inLength)
        {
            // 如果额外格式数据缓存足够容纳新进来的数据，则它们直接放入
            memcpy(mExtraBuffer + mExtraWritingOffset, inData, inLength);
            mExtraWritingOffset += inLength;
            return TRUE;
        }
        else
        {
            // 如果额外格式数据缓存不足以容纳新进来的数据,说明这一块数据包含额外格式
            // 数据和正常数据流数据两部分，则先将额外的格式数据放入专用缓存中
            memcpy(mExtraBuffer + mExtraWritingOffset, inData, space);
            mExtraWritingOffset += space;
            // 然后将剩下的数据（正常的数据流数据）递归调用AddData函数将它们放入
            // 正常的数据缓存
            return AddData(inData + space, inLength - space);
        }
    }

    // 将数据放入正常的、顺序接收的数据缓存（省略）
    // ……
```

```
}
// 设置数据流中的偏移量，准备下一次数据的（随机）读取
HRESULT CMediaController::SetPointer(LONGLONG llPos)
{
    // 仅在Filter连接时的格式校验过程支持SetPointer功能，正常数据流播放过程中的偏移
    // 计算由Read函数完成
    if (!mCheckingFinished && mReceiveFilter->IsStopped())
    {
        if (llPos >= 0 && llPos < mBufferSize)//llPos < mTotalSize)
        {
            CAutoLock   lck(&mSyncBuffer);
            mReadingOffset = long(llPos);
            // 下一次的数据（Read函数调用）从正常的缓存中读取
            mReadFromExtraBuffer = FALSE;
        }
    }
    // 如果当前的数据类型需要额外格式数据的支持，并且现在设置的偏移量超出了额外格式数
    // 据在整个流中的偏移量
    if (mExtraCheckOffset > 0 && llPos >= mExtraCheckOffset)
    {
        // 在额外格式数据缓存中定位
        mExtraReadingOffset  = long(llPos - mExtraCheckOffset);
        // 下一次的数据（Read函数调用）从额外格式数据的缓存中读取
        mReadFromExtraBuffer = TRUE;
    }
    return NOERROR;
}
// 外部调用者想要读取dwBytesToRead大小的数据到pbBuffer缓存中，Read函数返回时指定
// pdwBytesRead为实际读到的数据量
HRESULT CMediaController::Read(PBYTE pbBuffer, DWORD dwBytesToRead,
    BOOL bAlign, LPDWORD pdwBytesRead)
{
    // 如果要求读取额外的格式数据，则从保存额外格式数据的缓存中读取
    if (mExtraCheckOffset > 0 && mReceiveFilter->IsStopped() &&
        mReadFromExtraBuffer)
    {
        mReadFromExtraBuffer = FALSE; // 复位标记!
        memcpy(pbBuffer, mExtraBuffer + mExtraReadingOffset,
            dwBytesToRead);
        mExtraReadingOffset += dwBytesToRead;
        *pdwBytesRead   = dwBytesToRead;
        return NOERROR;
    }

    // 从正常的数据缓存中读取数据（省略）
    // ……
}
```

3.3.6 支持AVI

支持AVI原本是没有任何意义的，因为AVI文件的音视频一般不经过压缩，数据量很大，

不适合直接在网络上传输。但是，自从MPEG4算法得到实际应用之后，一些视频采用DivX、XVid等压缩算法的媒体文件也保存为AVI。这使得AVI文件拥有了新的生命力。单纯从文件大小上看，在网络上传输AVI数据似乎也未尝不可了。

小知识：AVI 文件格式

AVI（Audio Video Interleaved）是一种RIFF（Resource Interchange File Format）文件格式，多用于音视频捕捉、编辑、回放等应用程序中。通常情况下，一个AVI文件可以包含多个不同类型的媒体流（典型的情况下有一个音频流和一个视频流），不过含有单一音频流或单一视频流的AVI文件也是合法的。AVI可以算是Windows操作系统上最基本的、也是最常用的一种媒体文件格式。

先来介绍RIFF文件格式。RIFF文件使用四字符码FOURCC（four-character code）来表示数据类型，如‘RIFF’、‘AVI’、‘LIST’等。注意，Windows操作系统使用的字节顺序是little-endian，因此一个四字符码‘abcd’实际的DWORD值应为0x64636261。另外，四字符码中像‘AVI ’一样含有空格也是合法的。

RIFF文件首先含有一个如图3.31所示的文件头结构。

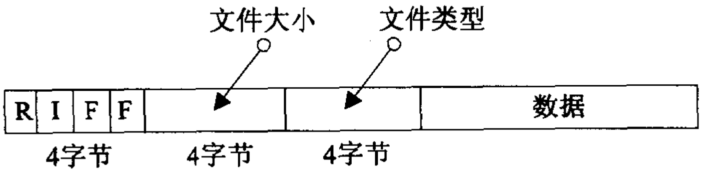


图 3.31　RIFF 文件结构

最开始的4个字节是一个四字符码‘RIFF’，表示这是一个RIFF文件；紧跟着后面用4个字节表示此RIFF文件的大小；然后又是一个四字符码说明文件的具体类型（如AVI、WAVE等）；最后就是实际的数据。注意文件大小值的计算方法为：实际数据长度 +4（文件类型域的大小）；也就是说，文件大小的值不包括‘RIFF’域和“文件大小”域本身的大小。

RIFF文件的实际数据中，通常还使用了列表（List）和块（Chunk）的形式来组织。列表可以嵌套子列表和块。其中，列表的结构为：**‘LIST’ listSize listType listData**——‘LIST’是一个四字符码，表示这是一个列表；listSize占用4字节，记录了整个列表的大小；listType也是一个四字符码，表示本列表的具体类型；listData就是实际的列表数据。注意listSize值的计算方法为：实际的列表数据长度 +4（listType域的大小）；也就是说listSize值不包括‘LIST’域和listSize域本身的大小。再来看块的结构：**ckID ckSize ckData**——ckID是一个表示块类型的四字符码；ckSize占用4字节，记录了整个块的大小；ckData为实际的块数据。注意ckSize值指的是实际的块数据长度，而不包括ckID域和ckSize域本身的大小。（注意：在下面的内容中，将以LIST (listType (listData))的形式来表示一个列表，以ckID (ckData)的形式来表示一个块，如[optional element]中括号中的元素表示为可选项。）

接下来介绍AVI文件格式。AVI文件类型用一个四字符码‘AVI ’来表示。整个AVI文件的结构为：一个RIFF头 + 两个列表（一个用于描述媒体流格式、一个用于保存媒体

流数据）+一个可选的索引块。AVI文件的展开结构大致如下:

```
RIFF ( 'AVI '
     LIST ( 'hdrl'
           'avih' (主AVI信息头数据)
          LIST ( 'strl'
                'strh'  (流的头信息数据)
                'strf'  (流的格式信息数据)
               [ 'strd'  (可选的额外的头信息数据) ]
               [ 'strn'  (可选的流的名字) ]
               ...
              )
           ...
         )
     LIST ( 'movi'
          { SubChunk | LIST ( 'rec '
                          SubChunk1
                          SubChunk2
                          ...
                         )
            ...
          }
          ...
         )
     [ 'idx1' (可选的AVI索引块数据) ]
    )
```

首先，RIFF（'AVI' ...）表征了AVI文件类型。然后就是AVI文件必需的第一个列表——'hdrl'列表，用于描述AVI文件中各个流的格式信息（AVI文件中的每一路媒体数据都称为一个流）。'hdrl'列表嵌套了一系列块和子列表——首先是一个'avih'块，用于记录AVI文件的全局信息，如流的数量、视频图像的宽和高等，可以使用一个AVIMAINHEADER数据结构来操作:

```
typedef struct _avimainheader {
    FOURCC fcc;       // 必须为 'avih'
    DWORD  cb;        // 本数据结构的大小，不包括最初的8个字节（fcc和cb两个域）
    DWORD  dwMicroSecPerFrame;      // 视频帧间隔时间（以毫秒为单位）
    DWORD  dwMaxBytesPerSec;        // 这个AVI文件的最大数据率
    DWORD  dwPaddingGranularity;    // 数据填充的粒度
    DWORD  dwFlags;                 // AVI文件的全局标记，比如是否含有索引块等
    DWORD  dwTotalFrames;     // 总帧数
    DWORD  dwInitialFrames;   // 为交互格式指定初始帧数（非交互格式应该指定为0）
    DWORD  dwStreams;         // 本文件包含的流的个数
    DWORD  dwSuggestedBufferSize; // 建议读取本文件的缓存大小（应能容纳最大的块）
    DWORD  dwWidth;           // 视频图像的宽（以像素为单位）
    DWORD  dwHeight;          // 视频图像的高（以像素为单位）
    DWORD  dwReserved[4];     // 保留
} AVIMAINHEADER;
```

然后，就是一个或多个'strl'子列表。（文件中有多少个流，这里就对应有多少个'strl'子列表。）每个'strl'子列表至少包含一个'strh'块和一个'strf'块，而'strd'块（保

存编解码器需要的一些配置信息）和‘strn’块（保存流的名字）是可选的。首先是‘strh’块，用于说明这个流的头信息，可以使用一个AVISTREAMHEADER数据结构来操作：

```
typedef struct _avistreamheader {
    FOURCC fcc;        // 必须为‘strh’
    DWORD cb;          // 本数据结构的大小，不包括最初的8个字节（fcc和cb两个域）
    FOURCC fccType;    // 流的类型：‘auds’（音频流）、‘vids’（视频流）、
                       // ‘mids’（MIDI流）、‘txts’（文字流）
    FOURCC fccHandler; // 指定流的处理者，对于音视频来说就是解码器
    DWORD  dwFlags;    // 标记：是否允许这个流输出？调色板是否变化？
    WORD   wPriority;  // 流的优先级（当有多个相同类型的流时优先级最高的为默认流）
    WORD   wLanguage;
    DWORD  dwInitialFrames;      // 为交互格式指定初始帧数
    DWORD  dwScale;              // 这个流使用的时间尺度
    DWORD  dwRate;
    DWORD  dwStart;    // 流的开始时间
    DWORD  dwLength;   // 流的长度（单位与dwScale和dwRate的定义有关）
    DWORD  dwSuggestedBufferSize;  // 读取这个流数据建议使用的缓存大小
    DWORD  dwQuality;              // 流数据的质量指标（0 ~ 10 000）
    DWORD  dwSampleSize;           // Sample的大小
    struct {
        short int left;
        short int top;
        short int right;
        short int bottom;
} rcFrame;      // 指定这个流（视频流或文字流）在视频主窗口中的显示位置
                // 视频主窗口由AVIMAINHEADER结构中的dwWidth和dwHeight决定
} AVISTREAMHEADER;
```

然后是‘strf’块，用于说明流的具体格式。如果是视频流，则使用一个BITMAPINFO数据结构来描述；如果是音频流，则使用一个WAVEFORMATEX数据结构来描述。

当AVI文件中的所有流都使用一个‘strl’子列表说明以后（注意：‘strl’子列表出现的顺序与媒体流的编号是对应的，如第一个‘strl’子列表说明的是第一个流（Stream 0），第二个‘strl’子列表说明的是第二个流（Stream 1），以此类推），‘hdrl’列表的任务也就完成了，随后跟着的就是AVI文件必需的第二个列表——‘movi’列表，用于保存真正的媒体流数据（视频图像帧数据或音频采样数据等）。那么，怎么来组织这些数据呢？可以将数据块直接嵌在‘movi’列表里面，也可以将几个数据块分组成一个‘rec’列表后再编排进‘movi’列表。（注意：在读取AVI文件内容时，建议将一个‘rec’列表中的所有数据块一次性读出。）但是，当AVI文件中包含有多个流的时候，数据块与数据块之间如何来区别呢？于是数据块使用了一个四字符码来表征它的类型，这个四字符码由2个字节的类型码和2个字节的流编号组成。标准的类型码定义如下：‘db’（非压缩视频帧）、‘dc’（压缩视频帧）、‘pc’（改用新的调色板）、‘wb’（音缩视频）。例如第一个流（Stream 0）是音频，则表征音频数据块的四字符码为‘00wb’；第二个流（Stream 1）是视频，则表征视频数据块的四字符码为‘00db’或‘00dc’。对于视频数据来说，在AVI数据序列中间还可以定义一个新的调色板，每个改变的调色板数据块用‘xxpc’来表征，新的调色板使用一个数据结构AVIPALCHANGE来定义。（注意：如果一个流的调色板中途可能改

变，则应在这个流格式的描述中，也就是AVISTREAMHEADER结构的dwFlags中包含一个AVISF_VIDEO_PALCHANGES标记。）另外，文字流数据块可以使用随意的类型码表征。

最后，紧跟在‘hdrl’列表和‘movi’列表之后的，就是AVI文件可选的索引块。这个索引块为AVI文件中每一个媒体数据块进行索引，并且记录它们在文件中的偏移（可能相对于‘movi’列表，也可能相对于AVI文件开头）。索引块使用一个四字符码‘idx1’来表征，索引信息使用一个数据结构AVIOLDINDEX来定义。

```
typedef struct _avioldindex {
   FOURCC  fcc;        // 必须为‘idx1’
   DWORD   cb;         // 本数据结构的大小，不包括最初的8个字节（fcc和cb两个域）
   struct _avioldindex_entry {
      DWORD   dwChunkId;      // 表征本数据块的四字符码
      DWORD   dwFlags;        // 说明本数据块是否是关键帧、‘rec ’列表等信息
      DWORD   dwOffset;       // 本数据块在文件中的偏移量
      DWORD   dwSize;         // 本数据块的大小
  } aIndex[]; // 这是一个数组！为每个媒体数据块都定义一个索引信息
} AVIOLDINDEX;
```

注意：如果一个AVI文件包含有索引块，则应在主AVI信息头的描述中，也就是AVIMAINHEADER结构的dwFlags中包含一个AVIF_HASINDEX标记。
还有一种特殊的数据块，用一个四字符码‘JUNK’来表征，它用于内部数据的对齐（填充），应用程序应该忽略这些数据块的实际意义。
上述关于AVI文件格式的介绍，并不包括OpenDML AVI M-JPEG文件格式小组制定的OpenDML AVI文件格式扩展部分的内容。相对该扩展部分有更多了解的读者，请参考“OpenDML AVI File Format Extensions”一文（此文可在Internet上搜索获得）。

如果一个AVI媒体节目中的视频采用了DivX、XVid等编码格式，则在客户机上播放这种节目之前安装这些格式的解码Filter是必需的。另外，为了保证Source Filter能够与AVI Splitter相连，Source Filter的输出Pin上应该使用媒体类型（MEDIATYPE_Stream，MEDIASUBTYPE_Avi）。不过，仅仅做出这些努力仍然是不够的。因为如果一个AVI文件在文件尾部含有索引块的话，AVI Splitter在与Source Filter连接的时候会要求读取这部分的数据。与支持MPEG2类似，服务器应该首先将这个索引块数据发送给客户机；否则Source Filter往下的Filter连接将无法成功。而对于客户机，则无需再作出代码上的改动——AVI的支持过程与MPEG2是一样的。相比之下，如何在服务器程序中找到一个AVI文件的索引块倒是关键（这部分实现将在3.3.7节中介绍）！

还有一点需要提醒读者的是：AVI毕竟不是流式的文件格式，如果一个AVI文件的数据块组织得不是很常规，则AVI Splitter可能会根据索引块提供的信息向Source Filter随机索取数据。这种数据读取的方式，对于Source Filter缓存数据的要求比较高，在HQ Net Receiver中并没有支持。因此，客户机上播放这种AVI媒体节目显示的视频图像可能会不正常。有一种可行的解决方案：将这种AVI文件在服务器上进行音视频分离以后再发送给客户机；客户机使用一个推模式的Source Filter接收网络数据，将接收到的音视频数据直接“推”给对应的解码Filter即可。

3.3.7 程序导读

首先来了解一下VODDemo项目的VC工作区结构，如图3.32所示。

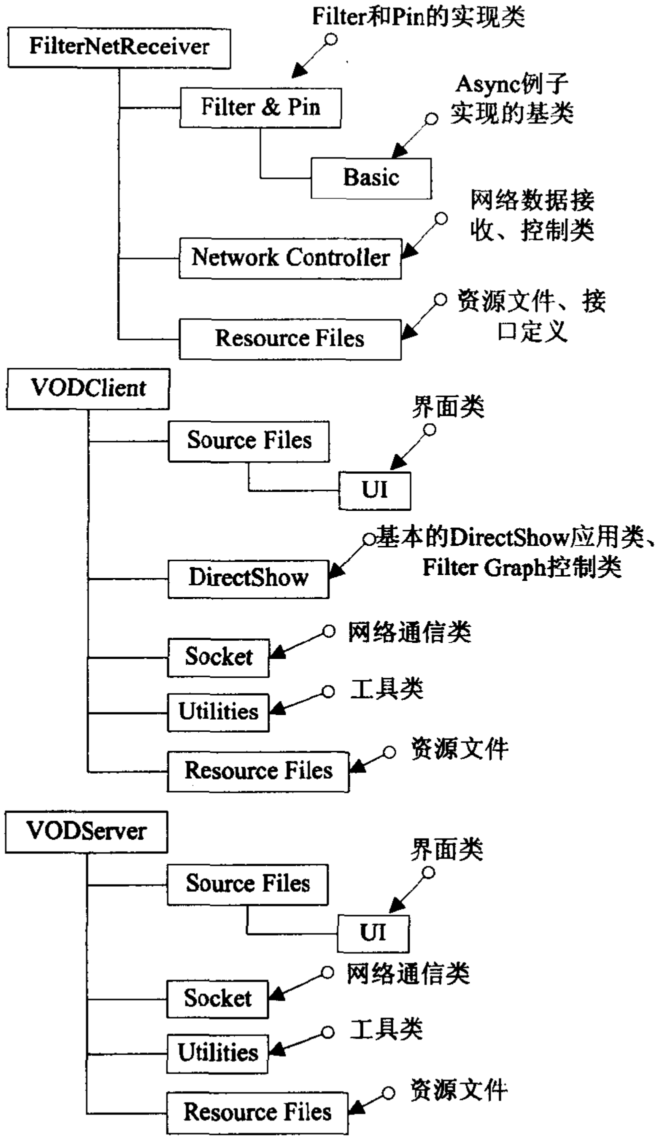


图 3.32　VODDemo 项目的 VC 工作区结构

VODDemo使用到的一些重要的类的继承结构如图3.33所示。

提示：VODDemo演示程序在实现时，较多地搭配使用了CMsgStation类与CMsgReceiver类，以实现一种自定义消息的传递机制。关于CMsgStation和CMsgReceiver两个类的功能介绍，请读者参考2.4.1节。

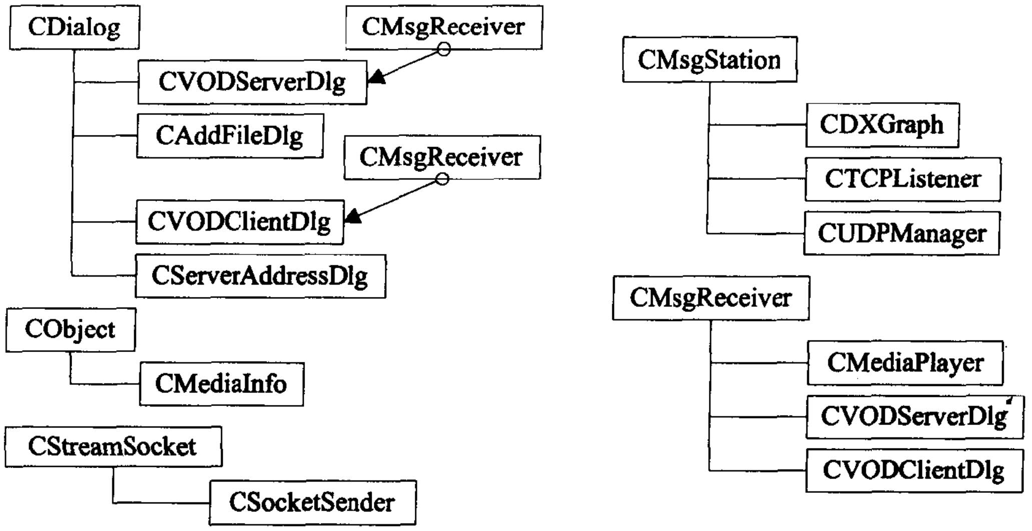


图 3.33 VODDemo 的类继承结构

接着来看一下服务器程序与客户机程序之间的交互过程。服务器与客户机之间的控制命令使用UDP协议传输，媒体数据使用TCP协议传输（侦听TCP连接的Socket放在客户机程序中实现）。VODDemo定义的UDP命令如下：

```
typedef enum
{
    Cmd_RequestTCPConnect = 0,  // 客户机要求建立TCP连接
    Cmd_RequestTCPDisconnect,   // 客户机要求断开TCP连接
    Cmd_RequestProgramList,     // 客户机要求得到节目列表
    Cmd_ProgramList,            // 服务器发送节目列表
    Cmd_RequestPlay,            // 要求服务器播放指定节目项
    Cmd_RequestPause,           // 要求服务器暂停指定节目项
    Cmd_RequestStop,            // 要求服务器停止播放指定节目项
} Net_Command;
```

其中Cmd_RequestTCPConnect命令带的参数使用如下的数据结构：

```
struct Request_TCP_Connect
{
    long client_ip;   // 客户机的IP地址
    WORD client_udp;  // 客户机的命令接收端口（UDP）
    WORD client_tcp;  // 客户机的TCP连接侦听端口
    // 将成员变量值从主机字节顺序转换成网络字节顺序
    void my_hton(void)
    {
        client_ip       = htonl(client_ip);
        client_udp      = htons(client_udp);
        client_tcp      = htons(client_tcp);
    };
    // 将成员变量值从网络字节顺序转换成主机字节顺序
    void my_ntoh(void)
    {
        client_ip       = ntohl(client_ip);
        client_udp      = ntohs(client_udp);
```

```
        client_tcp      = ntohs(client_tcp);
    };
};
```

Cmd_ProgramList命令带的参数使用如下的数据结构：

```
struct Program_List
{
    char file_path[100];     // 文件路径
    long pid;                // 媒体节目的惟一ID
    long file_type;          // 文件类型
    long file_size;          // 文件大小
    long check_size;         // 预接收数据量（用于Filter连接时的格式校验）
    long check_offset2;      // 额外的格式数据在文件中的偏移量（Filter连接时用到）
    // 将成员变量值从主机字节顺序转换成网络字节顺序
    void my_hton(void)
    {
        pid             = htonl(pid);
        file_type       = htonl(file_type);
        file_size       = htonl(file_size);
        check_size      = htonl(check_size);
        check_offset2   = htonl(check_offset2);
    };
    // 将成员变量值从网络字节顺序转换成主机字节顺序
    void my_ntoh(void)
    {
        pid             = ntohl(pid);
        file_type       = ntohl(file_type);
        file_size       = ntohl(file_size);
        check_size      = ntohl(check_size);
        check_offset2   = ntohl(check_offset2);
    };
};
```

媒体节目播放控制命令（包括Cmd_RequestPlay、Cmd_RequestPause、Cmd_RequestStop）带的参数使用如下的数据结构：

```
struct Media_Control
{
    long program_id;   // 指定要控制播放的媒体节目的ID
    // 将成员变量值从主机字节顺序转换成网络字节顺序
    void my_hton(void)
    {
        program_id  = htonl(program_id);
    };
    // 将成员变量值从网络字节顺序转换成主机字节顺序
    void my_ntoh(void)
    {
        program_id  = ntohl(program_id);
    };
};
```

客户机与服务器建立连接的过程如图3.34所示。

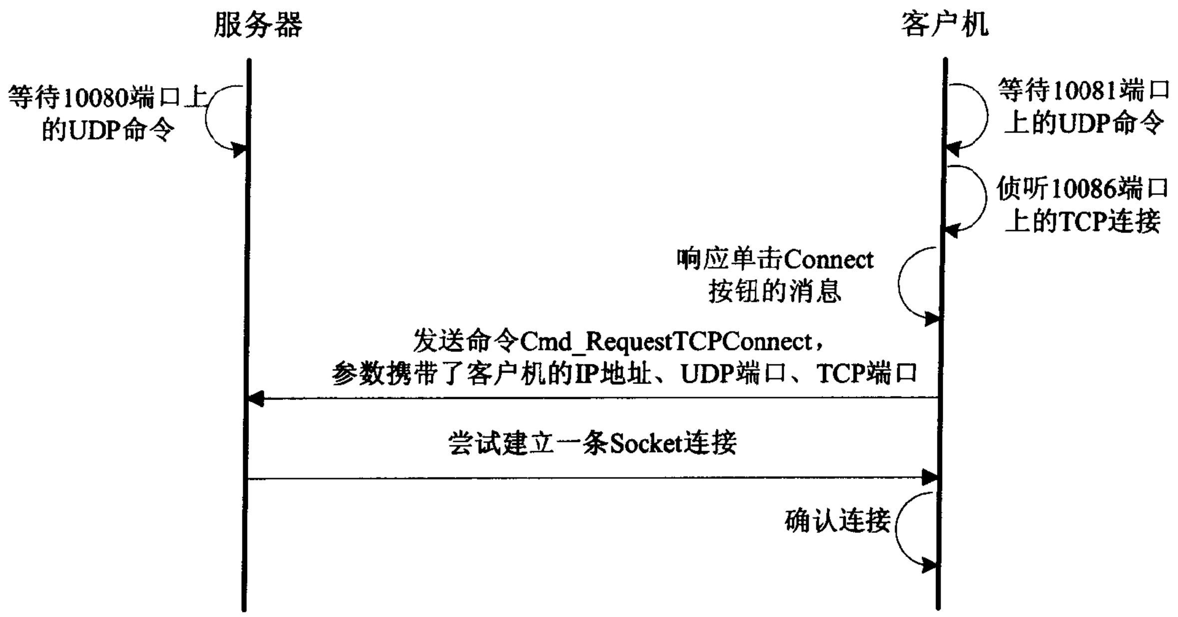


图 3.34　客户机与服务器建立连接的过程

图3.34中的过程在程序中的实现如下：

```
/////////////////////// 客户机部分//////////////////////////
// VODClientDlg.cpp : implementation file
//
// Connect按钮的响应函数
void CVODClientDlg::OnButtonConnect()
{
    // 弹出一个如图3.24所示的对话框让用户指定服务器的IP地址
    CServerAddressDlg dlg;
    dlg.SetServerIP(&mServerIP);
    if (dlg.DoModal() == IDOK)
    {
        // 创建发送UDP命令用的Socket
        mCommandManager.SetTargetIP(mServerIP);
        mCommandManager.SetTargetPort(SERVER_UDP_PORT);
        if (mCommandManager.CreateSender())
        {
            // 发送一个要求建立TCP连接的命令
            BOOL pass = SendTCPConnectRequest();
        }
    }
}
// 发送一个要求建立TCP连接的命令
BOOL CVODClientDlg::SendTCPConnectRequest(void)
{
    char buf[MAX_COMMAND_SIZE];          // 命令数据缓存
    long len = sizeof(Net_Command) + sizeof(Request_TCP_Connect);
    long * pCmd = (long *) buf;
    *pCmd = Cmd_RequestTCPConnect;       // 指定命令类型
    *pCmd = htonl(*pCmd);                // 转换字节顺序
```

```
    pCmd++;
    // 指定命令的参数
    Request_TCP_Connect * pData = (Request_TCP_Connect *) pCmd;
    pData->client_ip    = mClientIP;
    pData->client_tcp   = CLIENT_TCP_PORT;
    pData->client_udp   = CLIENT_UDP_PORT;
    pData->my_hton();
    // 向服务器发送命令
    return mCommandManager.Send(buf, len);
}

/////////////////////// 服务器部分//////////////////////////
//
// CUDPManager.cpp
//
// 服务器程序接收UDP命令的（线程）循环过程
void CUDPManager::ReceivingLoop(void)
{
    struct sockaddr_in  addr_cli;
    int  addr_cli_len = sizeof(addr_cli);
    char buffer[MAX_COMMAND_SIZE];        // 接收命令使用的缓存
    long bytes = 0;

    mIsReceiving = TRUE;
    while (mIsReceiving)
    {
        int addr_cli_len = sizeof(addr_cli);
        bytes = recvfrom(mSckReceiver, (char *)buffer, MAX_COMMAND_SIZE,
            0, (LPSOCKADDR) &addr_cli, (int *) &addr_cli_len);
        if (bytes == SOCKET_ERROR || bytes == 0)
        {
            mIsReceiving = FALSE;
        }
        else
        {
            // 接收到命令后发送给上层观察者处理
            Broadcast(msg_CommandReceived, buffer);
        }
    }
}
// VODServerDlg.cpp : implementation file
//
bool CVODServerDlg::ReceiveMessage(MessageT inMessage, void * ioParam, void
    * ioParam2)
{
    if (inMessage == msg_CommandReceived)
    {
        ProcessCommand(ioParam);
        return true;
    }
    return CMsgReceiver::ReceiveMessage(inMessage, ioParam, ioParam2);
}
```

```
// 处理各种命令
void CVODServerDlg::ProcessCommand(void * inCommand)
{
    // 得到命令的类型
    long * pCmd = (long *) inCommand;
    switch (ntohl(*pCmd))    // 注意字节顺序转换
    {
    case Cmd_RequestTCPConnect:
        {
            pCmd++;  // 指向命令参数
            // 使用Request_TCP_Connect数据结构操作命令参数
            Request_TCP_Connect * pData = (Request_TCP_Connect *) pCmd;
            pData->my_ntoh();  // 转换字节顺序
            // 保存客户机的IP地址、命令端口号
            mClientIP       = pData->client_ip;
            mClientCmdPort  = pData->client_udp;
            // 尝试与客户机建立TCP连接
            mNetSender.ConnectTo(pData->client_ip, pData->client_tcp);
        }
        break;

    // 其他命令处理（省略）
    // ……
    }
}

////////////////////// 客户机部分/////////////////////////
//
// CTCPListener.cpp
//
// 客户机程序侦听TCP连接的线程执行体
DWORD WINAPI CTCPListener::ListeningThrd(void *pParam)
{
    ASSERT(pParam);

    CTCPListener * pListen = (CTCPListener *) pParam;
    pListen->mIsListening  = TRUE;
    while (pListen->mIsListening)
    {
        if (!pListen->Accept())
        {
            pListen->mIsListening = FALSE;
            break;
        }
        else
        {
            // 当接受一个TCP连接后通知上层观察者
            pListen->Broadcast(msg_NewSocketAccepted);
        }
    }

    return 1;
```

```
}
// VODClientDlg.cpp : implementation file
//
bool CVODClientDlg::ReceiveMessage(MessageT inMessage, void * ioParam, void
    * ioParam2)
{
    if (inMessage == msg_CommandReceived)
    {
        ProcessCommand(ioParam);
        return true;
    }
    else if (inMessage == msg_NewSocketAccepted)
    {
        // 当TCP连接建立后刷新内部状态为“已连接”
        mState = CONNECTED;
        // 更新界面上的状态显示
        ReflectStatus();
        return true;
    }
    return CMsgReceiver::ReceiveMessage(inMessage, ioParam, ioParam2);
}
```

当客户机与服务器成功建立连接之后,用户可以单击客户机程序界面上的Refresh按钮,以获得服务器上供点播的节目列表。此时,客户机与服务器之间的交互过程如图3.35所示。

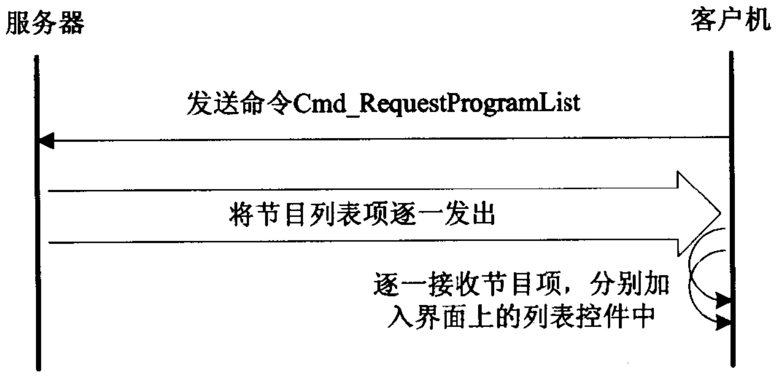


图 3.35　客户机获取节目列表的过程

图3.35中的过程在程序中的实现如下:

```
////////////////////// 客户机部分///////////////////////////
// VODClientDlg.cpp : implementation file
//
// Refresh按钮的响应函数
void CVODClientDlg::OnButtonRefresh()
{
    mProgramListCtrl.DeleteAllItems();
    SendSimpleCommand(Cmd_RequestProgramList);
}
// 发送无参数的简单命令
BOOL CVODClientDlg::SendSimpleCommand(long inCommand)
```

```
{
    char  buf[MAX_COMMAND_SIZE];
    long  len  = sizeof(Net_Command);
    long * pCmd = (long *) buf;
    *pCmd = inCommand;       // 指定命令类型
    *pCmd = htonl(*pCmd);    // 转换字节顺序
    return mCommandManager.Send(buf, len);
}

////////////////////// 服务器部分/////////////////////////
// VODServerDlg.cpp : implementation file
//
void CVODServerDlg::ProcessCommand(void * inCommand)
{
    long * pCmd = (long *) inCommand;
    switch (ntohl(*pCmd))
    {
    case Cmd_RequestProgramList:
        SendProgramList();
        break;

    // 其他命令处理（省略）
    // ……
    }
}
// 将服务器上的所有节目列表项发送给客户机
void CVODServerDlg::SendProgramList(void)
{
    // 创建向客户机发送UDP命令用的Socket
    mCommandManager.SetTargetIP(mClientIP);
    mCommandManager.SetTargetPort(mClientCmdPort);
    if (mCommandManager.CreateSender())
    {
        char buffer[MAX_COMMAND_SIZE];
        long  len  = sizeof(Net_Command) + sizeof(Program_List);
        long * pCmd = (long *) buffer;
        *pCmd = Cmd_ProgramList;    // 指定命令类型
        *pCmd = htonl(*pCmd);       // 转换字节顺序
        pCmd++;                     // 指向参数区
        Program_List * pData = (Program_List *) pCmd;
        POSITION pos = mPrgrmList.GetHeadPosition();
        while (pos)
        {
            // 逐一获得节目项
            CMediaInfo info = mPrgrmList.GetNext(pos);
            info.CopyTo(pData);
            pData->my_hton();   // 转换字节顺序
            // 发送当前节目项
            mCommandManager.Send(buffer, len);
        }
    }
}
```

```
/////////////////////// 客户机部分//////////////////////////
// VODClientDlg.cpp : implementation file
//
void CVODClientDlg::ProcessCommand(void * inCommand)
{
    long * pCmd = (long *) inCommand;
    switch (ntohl(*pCmd))
    {
    case Cmd_ProgramList:
        {
            pCmd++;
            Program_List * pData = (Program_List *) pCmd;
            pData->my_ntoh();  // 转换字节顺序
            // 保存接收到的节目项信息
            CMediaInfo info;
            info.CopyFrom(pData);
            AddProgramItem(info);
        }
        break;

    default:
        break;
    }
}
// 将一个节目项信息加入到界面上的列表控件中显示，同时保存到一个列表对象中
// 节目项使用CMediaInfo类描述
void CVODClientDlg::AddProgramItem(CMediaInfo & inMedia)
{
    // 将节目项加入到界面上的列表控件中显示
    int count = mProgramListCtrl.GetItemCount();
    char * p = strrchr(inMedia.mFilePath, '\\');
    mProgramListCtrl.InsertItem(count, p+1);
    CString str;
    switch (inMedia.mFileType)
    {
    case FT_MPEG1:
        str = "MPEG1";
        break;
    case FT_MPEG2:
        str = "MPEG2";
        break;
    case FT_AVI:
        str = "AVI";
        break;
    case FT_MP3:
        str = "MP3";
        break;
    default:
        str = "Unknown";
        break;
    }
    mProgramListCtrl.SetItemText(count, 1, str);
```

```
        str.Format("%.3fM", inMedia.mFileSize / 1024. / 1024.);
        mProgramListCtrl.SetItemText(count, 2, str);
        str.Format("%d", inMedia.mProgramId);
        mProgramListCtrl.SetItemText(count, 3, str);
        // 将节目项同时放到一个列表中保存
        mPrgrmList.AddTail(inMedia);
    }
```

当客户机从服务器获得节目列表后，可以指定一个感兴趣的节目项进行播放。节目项在客户机和服务器之间使用一个惟一数字代号（PID）来表示。此时，客户机与服务器之间的交互过程如图3.36所示。

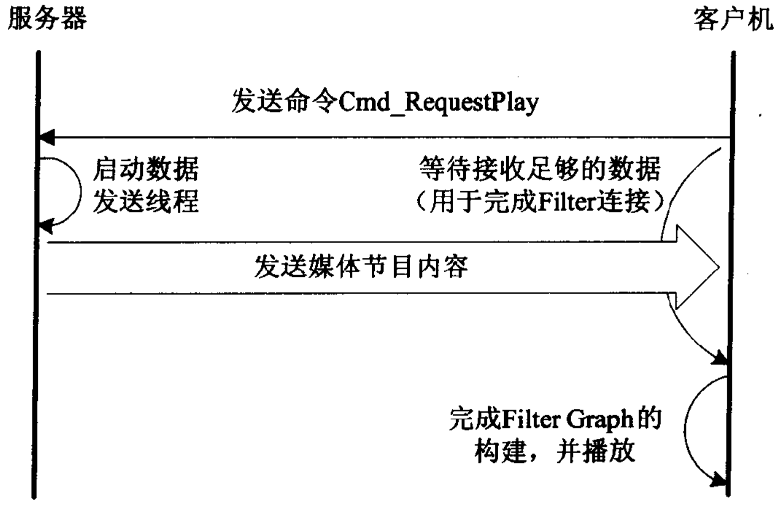


图 3.36　客户机播放指定的节目项的过程

图3.36中的过程在程序中的实现如下：

```
//////////////////////// 客户机部分///////////////////////////
// VODClientDlg.cpp : implementation file
//
// Play按钮的响应函数
void CVODClientDlg::OnButtonPlay()
{
    // 如果当前状态是“暂停”，只需向服务器再次发送一个播放命令
    if (mState == PAUSE)
    {
        SendMediaControlCommand(Cmd_RequestPlay);
        mPlayer.PlayGraph();
        mState = PLAY;
        ReflectStatus();
        return;
    }
    // 如果是第一次对指定的节目项执行播放命令，则需构建Filter Graph，首先得到用户选
    // 择的节目项
    int index = mProgramListCtrl.GetNextItem(-1, LVNI_SELECTED);
    if (index >= 0)
    {
        // 如果此时还没建立TCP连接，则向服务器发送建立TCP连接的命令
```

```
if (!mTCPListener.IsAcceptedValid())
{
    SendTCPConnectRequest();
}
// 得到用户选择的节目项的PID
CString pid = mProgramListCtrl.GetItemText(index, 3);
mPlayingPID = atol(pid);
// 得到该节目项的完整描述信息，然后设置给控制对象
CMediaInfo    media;
POSITION pos = mPrgrmList.GetHeadPosition();
while (pos)
{
    media  = mPrgrmList.GetNext(pos);
    if (media.mProgramId == mPlayingPID)
    {
        break;
    }
}
// mPlayer是控制类CMediaPlayer的一个实例
mPlayer.SetMediaInfo(media);
// 开始构建Filter Graph（并没有完全连接）
BOOL pass = mPlayer.Activate();
if (pass)
{
    mPlayer.SetStreamType(media.mFileType);
    mPlayer.SetSize(media.mFileSize, media.mCheckOffset2);
    if (!mTCPListener.IsAcceptedValid())
    {
        Sleep(500);
    }
    mPlayer.SetStreamSocket(mTCPListener.GetAccepted());
    // 启动Source Filter的数据接收线程
    mPlayer.StartReceiving();
}
// 向服务器发送要求播放的命令
if (pass)
{
    pass = SendMediaControlCommand(Cmd_RequestPlay);
}
if (pass)
{
    BOOL  canConnect = FALSE;
    long  counter    = 0;
    // 等待接收足够的数据
    while (!canConnect)
    {
        // 询问Source Filter是否接收到了足够的数据，然后才可以将Source
        // Filter往下与其他Filter连接
        if (mPlayer.CanConnectToDownstream())
        {
            canConnect = TRUE;
        }
```

```
                else
                {
                    Sleep(100);
                    if (++counter > 200)
                    {
                        break;
                    }
                }
            }
            if (canConnect) // 接收到了足够的数据
            {
                // 开始连接Filter，以构建完整的播放用的Filter Graph
                pass = mPlayer.ConnectFilters();
                // 运行Filter Graph
                pass = mPlayer.PlayGraph();
                // 更新内部状态
                mState = PLAY;
                ReflectStatus();
            }
        }
        mProgramListCtrl.SetFocus();
    }
    else
    {
        AfxMessageBox("Please select a program first!");
    }
}
// 发送播放控制命令
BOOL CVODClientDlg::SendMediaControlCommand(long inCommand)
{
    char  buf[MAX_COMMAND_SIZE];
    long  len  = sizeof(Net_Command) + sizeof(Media_Control);
    long * pCmd = (long *) buf;
    *pCmd = inCommand;     // 指定命令类型
    *pCmd = htonl(*pCmd);  // 转换字节顺序
    pCmd++;  // 指向参数区
    Media_Control * pData = (Media_Control *) pCmd;
    pData->program_id = mPlayingPID;  // 指定媒体节目的PID
    pData->my_hton();  // 转换字节顺序
    return mCommandManager.Send(buf, len);  // 发送
}
//
// CMediaPlayer.cpp
//
// 询问Source Filter是否已经接收到了足够的数据（可以让Source Filter与后面的Filter
// 连接了）
BOOL CMediaPlayer::CanConnectToDownstream(void)
{
    HRESULT hr = S_FALSE;
    if (mINetReceiver)
    {
        hr = mINetReceiver->CanConnectToDownstream();
```

```
    }
    return (hr == S_OK);
}

/////////////////////// 服务器部分//////////////////////////
// VODServerDlg.cpp : implementation file
//
void CVODServerDlg::ProcessCommand(void * inCommand)
{
    long * pCmd = (long *) inCommand;
    switch (ntohl(*pCmd))
    {
    case Cmd_RequestPlay:
        {
            pCmd++;
            Media_Control * pData = (Media_Control *) pCmd;
            pData->my_ntoh();
            // 根据指定节目的PID，从列表中找个这个节目的完整描述
            POSITION pos = mPrgrmList.GetHeadPosition();
            while (pos)
            {
                CMediaInfo info = mPrgrmList.GetNext(pos);
                if (info.mProgramId == pData->program_id)
                {
                    // 启动数据发送线程发送媒体节目内容
                    mNetSender.SetSource(info);
                    mNetSender.SetPaused(FALSE);
                    mNetSender.StartSending();
                    break;
                }
            }
        }
        break;

    // 其他命令处理（省略）
    // ……
    }
}
//
// CSocketSender.cpp
//
void CSocketSender::SetSource(CMediaInfo & inMedia)
{
    mSource = inMedia;
    // 打开媒体文件
    if (mFile)
    {
        fclose(mFile);
        mFile = NULL;
    }
    mFile = fopen(mSource.mFilePath, "rb");
    ASSERT(mFile);
```

```
}
// 设置暂停标记（暂停时不向客户机发送数据）
void CSocketSender::SetPaused(BOOL inPaused)
{
    mPaused = inPaused;
}
// 数据发送线程中的发送循环过程
void CSocketSender::SendingLoop(void)
{
    // 如果客户机程序播放这种媒体类型需要额外的格式数据，则首先将这部分额外的格式数据
    // 发送出去
    if (mSource.mCheckOffset2 > 0)
    {
        // 在媒体文件中定位额外的格式数据
        long len = mSource.mFileSize - mSource.mCheckOffset2;
        char * pExtra = new char[len];
        fseek(mFile, mSource.mCheckOffset2, SEEK_SET);
        int bytes = fread(pExtra, sizeof(char), len, mFile);
        // 将文件重新定位到开头，准备发送正常的数据流
        fseek(mFile, 0, SEEK_SET);
        // 发送额外的格式数据
        bytes = send(mSocket, pExtra, len, 0);
        delete[] pExtra;
    }
    // 开始正常的数据流发送过程
    // const long PACK_SIZE = 2048;
    char  buf[PACK_SIZE]; // 发送数据使用的缓存
    int   readLen = 0;
    int   sentLen = 0;
    while (mIsSending)
    {
        if (mPaused)
        {
            Sleep(10);
            continue;
        }
        // 从文件中读取一块数据准备发送
        readLen = fread(buf, sizeof(char), PACK_SIZE, mFile);
        if (readLen > 0)
        {
            // 将数据发送给客户机
            sentLen = send(mSocket, buf, readLen, 0);
            if (sentLen == SOCKET_ERROR)
            {
                Detach(); // Close the socket
                mIsSending = FALSE;
                break;
            }
            // 如果从文件中读取的数据少于要求的数据量，可能是读文件发生错误或达到了文
            // 件的末尾
            if (readLen < PACK_SIZE)
            {
```

```
                mIsSending = FALSE;
                break;
            }
        }
        else
        {
            mIsSending = FALSE;
        }
    }
}
```

对指定媒体节目进行的播放控制还有“暂停”和“停止”。这两个过程都比较简单：“暂停”的实现，只需将客户机程序的Filter Graph暂停运行就可以了；而“停止”需要发送一个命令告诉服务器，让它不要再发送数据了。“停止”实现的过程如图3.37所示。

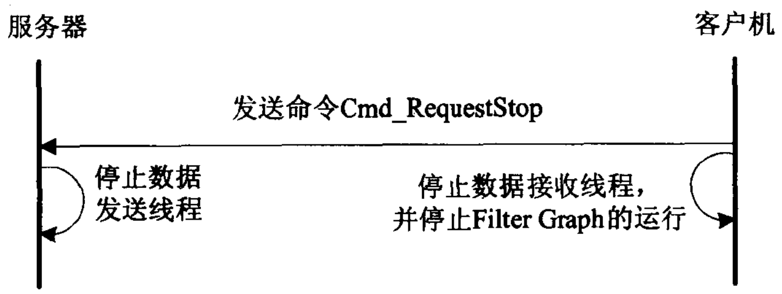


图 3.37 客户机停止播放指定的节目项的过程

“暂停”和“停止”两个控制过程在程序中的实现如下：

```
/////////////////////// 客户机部分//////////////////////////
// VODClientDlg.cpp : implementation file
//
// Pause按钮的响应函数
void CVODClientDlg::OnButtonPause()
{
    // 只需将Filter Graph暂停运行就可以了
    mPlayer.PauseGraph();
    // 更新内部状态
    mState = PAUSE;
    ReflectStatus();
    mProgramListCtrl.SetFocus();
}
// Stop按钮的响应函数
void CVODClientDlg::OnButtonStop()
{
    // 向服务器发送停止播放的命令
    SendMediaControlCommand(Cmd_RequestStop);
    // 停止本地Filter Graph的运行
    mPlayer.StopGraph();
    // 让Source Filter停止数据接收线程（会将TCP连接断开！）
    mPlayer.StopReceiving();
    // 更新内部状态
```

```
        mState = STOP;
        ReflectStatus();
        mProgramListCtrl.SetFocus();
    }

    ////////////////////// 服务器部分//////////////////////////
    // VODServerDlg.cpp : implementation file
    //
    void CVODServerDlg::ProcessCommand(void * inCommand)
    {
        long * pCmd = (long *) inCommand;
        switch (ntohl(*pCmd))
        {
        case Cmd_RequestPause:
            mNetSender.SetPaused(TRUE);  // 设置暂停标记
            break;

        case Cmd_RequestStop:
            // 停止数据发送线程（会将TCP连接断开！）
            mNetSender.SetPaused(FALSE);
            mNetSender.StopSending();
            break;

        // 其他命令处理（省略）
        // ……
        }
    }
```

看到这里，读者应该对VODDemo的客户机程序与服务器程序之间的交互过程以及具体实现大致有了了解。最后，我们还要来总结一下这两部分程序的关键实现。两部分程序的界面上都使用了一个列表控件来显示节目列表，并且主界面类中都使用了一个CList<CMediaInfo,CMediaInfo&>类型的列表对象来保存各节目项的详细信息。节目项的详细信息用CMediaInfo类来描述。对于服务器程序来说，当管理员加入一个媒体文件后，如何来识别媒体类型、获得媒体文件相关信息、特别是获取额外格式数据的描述至关重要。

```
//
// CMediaInfo.h
//

#ifndef __H_CMediaInfo__
#define __H_CMediaInfo__

#include <dshow.h>

#define MAX_FILE_PATH 100

class CMediaInfo : public CObject
{
public:
    char     mFilePath[MAX_FILE_PATH];   // 媒体文件的完整路径
    long     mProgramId;                 // 节目的惟一PID
```

```
    long    mFileType;          // 媒体文件类型
    long    mFileSize;          // 文件大小
    long    mCheckSize;         // 用于格式校验的数据量
    long    mCheckOffset2;      // 额外的格式数据在文件中的偏移量
public:
    CMediaInfo();
    CMediaInfo(const CMediaInfo&);  // 拷贝构造函数
    ~CMediaInfo();
    // 使用MFC的串行化存储节目列表
    DECLARE_SERIAL(CMediaInfo)
    void Serialize(CArchive& ar);
    // 赋值操作符重载
    CMediaInfo& operator=(const CMediaInfo&);
    // "克隆"数据成员
    void Clone(const CMediaInfo&);
    // 设置文件路径
    BOOL SetFilePath(const char * inFile);
    // 指定一个PID
    void SetProgramId(long inPID);
    // 将数据成员的值复制到Program_List数据结构的对应成员中
    void CopyTo(struct Program_List * outInfo);
    // 将Program_List数据结构各成员的值复制到本对象的对应数据成员中
    void CopyFrom(struct Program_List * inInfo);
private:
    // 分析文件
    BOOL ParseFile(const char * inFile);
    // 检查MPEG文件(MPEG1? MPEG2? )
    void CheckMpegFile(const char * inFile);
    IPin * GetPinByDirection(IBaseFilter * inFilter,
        PIN_DIRECTION inDirection);
    // 检查AVI文件(获取文件末尾的索引块偏移量)
    void CheckAviFile(const char * inFile);
};

#endif // __H_CMediaInfo__

//
// CMediaInfo.cpp
//

#include "stdafx.h"
#include "CMediaInfo.h"
#include "defines.h"
#include "CAviParser.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

IMPLEMENT_SERIAL(CMediaInfo, CObject, 1)  // 串行化实现
```

```
////////////////////////////////////////////////////////////////////////////
CMediaInfo::CMediaInfo()
{
    // 参数初始化
    mFilePath[0]    = '\0';
    mProgramId      = 0;
    mFileType       = 0;
    mFileSize       = 0;
    mCheckSize      = 10240;    // 10K
    mCheckOffset2   = 0;        // 默认情况下不需要额外的格式数据
}
// 拷贝构造函数
CMediaInfo::CMediaInfo(const CMediaInfo& inMedia)
{
    Clone(inMedia);
}
CMediaInfo::~CMediaInfo()
{
}
// 串行化存储
void CMediaInfo::Serialize(CArchive& ar)
{
    CObject::Serialize(ar);

    CString str;
    if (ar.IsStoring())
    {
        str = mFilePath;
        ar << str;
        ar << mProgramId;
        ar << mFileType;
        ar << mFileSize;
        ar << mCheckSize;
        ar << mCheckOffset2;
    }
    else
    {
        ar >> str;
        strcpy(mFilePath, str);
        ar >> mProgramId;
        ar >> mFileType;
        ar >> mFileSize;
        ar >> mCheckSize;
        ar >> mCheckOffset2;
    }
}
// 赋值操作符重载
CMediaInfo& CMediaInfo::operator=(const CMediaInfo& inMedia)
{
    Clone(inMedia);
    return *this;
}
```

```
// 克隆各数据成员的值
void CMediaInfo::Clone(const CMediaInfo& inMedia)
{
    strcpy(mFilePath, inMedia.mFilePath);
    mProgramId      = inMedia.mProgramId;
    mFileType       = inMedia.mFileType;
    mFileSize       = inMedia.mFileSize;
    mCheckSize      = inMedia.mCheckSize;
    mCheckOffset2   = inMedia.mCheckOffset2;
}
// CMediaInfo类的数据成员与Program_List数据结构的各成员对应转换
void CMediaInfo::CopyTo(struct Program_List * outInfo)
{
    memcpy(outInfo->file_path, mFilePath, MAX_FILE_PATH);
    outInfo->pid            = mProgramId;
    outInfo->file_type      = mFileType;
    outInfo->file_size      = mFileSize;
    outInfo->check_size     = mCheckSize;
    outInfo->check_offset2  = mCheckOffset2;
}
void CMediaInfo::CopyFrom(struct Program_List * inInfo)
{
    memcpy(mFilePath, inInfo->file_path, MAX_FILE_PATH);
    mProgramId      = inInfo->pid;
    mFileType       = inInfo->file_type;
    mFileSize       = inInfo->file_size;
    mCheckSize      = inInfo->check_size;
    mCheckOffset2   = inInfo->check_offset2;
}
// 设置文件路径
BOOL CMediaInfo::SetFilePath(const char * inFile)
{
    // 对文件进行分析（获取文件类型、文件大小、额外格式数据的偏移量等）
    if (ParseFile(inFile))
    {
        strcpy(mFilePath, inFile);
        return TRUE;
    }
    return FALSE;
}
// 指定一个PID
void CMediaInfo::SetProgramId(long inPID)
{
    mProgramId = inPID;
}
// 分析文件
BOOL CMediaInfo::ParseFile(const char * inFile)
{
    // 得到文件大小
    mFileSize = 0;
    CFileStatus  fileStatus;
    if (CFile::GetStatus(inFile, fileStatus))
```

```
    {
        mFileSize = fileStatus.m_size;
    }
    // 根据文件的扩展名初步判断文件的类型
    CString strExt = strrchr(inFile, '.');
    if (strExt.CompareNoCase(".mpeg") == 0 ||
        strExt.CompareNoCase(".mpg") == 0 ||
        strExt.CompareNoCase(".vob") == 0)
    {
        // 进一步检查这个MPEG文件是MPEG1还是MPEG2
        CheckMpegFile(inFile);
    }
    else if (strExt.CompareNoCase(".avi") == 0)
    {
        // 进一步获得AVI文件的索引块的偏移量
        CheckAviFile(inFile);
    }
    else if (strExt.CompareNoCase(".mp3") == 0)
    {
        // 这是一个MP3文件
        mFileType = FT_MP3;
    }
    else
    {
        AfxMessageBox("This file type is not supported!");
        return FALSE;
    }
    return TRUE;
}
// 判断一个MPEG文件是MPEG1还是MPEG2
// 这是一个很巧妙的方法（不需要知道MPEG格式的专业知识）：将指定该文件的File Source
// Filter与MPEG-1 Stream Splitter试连接，如果能连上，说明这是一个MPEG1文件；否则
// 可能是一个MPEG2文件。
void CMediaInfo::CheckMpegFile(const char * inFile)
{
    CoInitialize(NULL);  // COM库的初始化

    mFileType = FT_MPEG1;       // 默认为MPEG1
    IGraphBuilder * pBuilder = NULL;
    // 创建一个Filter Graph
    CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER,
        IID_IGraphBuilder, (void **) &pBuilder);
    if (pBuilder)
    {
        // 为指定的文件创建一个Source Filter
        IBaseFilter * pSource = NULL;
        WCHAR  wcFilePath[MAX_PATH];
        MultiByteToWideChar(CP_ACP, 0, inFile, -1, wcFilePath, MAX_PATH);
        pBuilder->AddSourceFilter(wcFilePath, L"Source", &pSource);
        pSource->Release();
        // 创建DirectShow标准的MPEG-1 Stream Splitter
        IBaseFilter * pSplitter = NULL;
```

```
        CoCreateInstance(CLSID_MPEG1Splitter, NULL, CLSCTX_INPROC_SERVER,
            IID_IBaseFilter, (void **)&pSplitter);
        pBuilder->AddFilter(pSplitter, L"MPEG-1 Splitter");
        pSplitter->Release();
        // 获得Source Filter的输出Pin
        IPin * pOut = GetPinByDirection(pSource, PINDIR_OUTPUT);
        // 获得MPEG-1 Stream Splitter的输入Pin
        IPin * pIn  = GetPinByDirection(pSplitter, PINDIR_INPUT);
        // 将Source Filter与MPEG-1 Stream Splitter试连接
        HRESULT hr  = pBuilder->ConnectDirect(pOut, pIn, NULL);
        if (FAILED(hr))
        {
            // 连接失败，认为这是一个MPEG2文件
            mFileType = FT_MPEG2;
            // 指定这个MPEG2文件末尾的1M数据作为额外的格式数据
            mCheckOffset2 = mFileSize - 1048576;
        }
        pBuilder->RemoveFilter(pSplitter);
        pBuilder->RemoveFilter(pSource);
        pBuilder->Release();
    }

    CoUninitialize();  // COM库的反初始化
}
// 工具函数：根据Pin的方向来获取Pin对象的指针
IPin * CMediaInfo::GetPinByDirection(IBaseFilter * inFilter, PIN_DIRECTION
    inDirection)
{
    IPin * foundPin = NULL;
    if (inFilter)
    {
        IEnumPins * pinEnum = 0;
        // 枚举Filter上的所有Pin
        if (SUCCEEDED(inFilter->EnumPins(&pinEnum)))
        {
            pinEnum->Reset();

            IPin * pin = 0;
            ULONG fetchCount = 0;
            while (!foundPin && SUCCEEDED(pinEnum->Next(1, &pin,
                &fetchCount)) && fetchCount)
            {
                if (pin)
                {
                    // 获取Pin的描述信息
                    PIN_INFO pinInfo;
                    if (SUCCEEDED(pin->QueryPinInfo(&pinInfo)))
                    {
                        pinInfo.pFilter->Release();
                        // Pin的方向是否符合要求
                        if (pinInfo.dir == inDirection)
                        {
```

```
                        foundPin = pin;
                    }
                }
                // 如果不是方向匹配的Pin，则将它释放
                if (!foundPin)
                {
                    pin->Release();
                }
                pin = 0;
            }
        }
        pinEnum->Release();
    }
}
// 返回的Pin对象不增加额外的引用计数，即外部在使用完这个Pin对象指针后不用再调用
// Release函数
if (foundPin)
{
    foundPin->Release();
}
return foundPin;
}
// 检查AVI文件（获取AVI文件尾部的索引块的偏移量）
void CMediaInfo::CheckAviFile(const char * inFile)
{
    // 使用CAviParser类对AVI文件进行格式分析
    CAviParser parser;
    if (parser.SetAviFile(inFile))
    {
        // 指定AVI文件类型
        mFileType = FT_AVI;
        // 开始分析AVI文件内容
        parser.Parse();
        // 获取AVI文件尾部索引块的偏移量
        mCheckOffset2 = (long) parser.m_idx1Pos;
    }
}
```

上述CMediaInfo类的实现中，对AVI文件进行的分析也是个关键。AVI文件的分析通过CAviParser类来完成，理解这个类的实现过程请读者同时参考3.3.6节。

```
//
// CAviParser.h
//

#ifndef __H_CAviParser__
#define __H_CAviParser__

#include <afxtempl.h>
#include <dshow.h>
#include <Aviriff.h>
```

```
class CAviParser
{
private:
    char    mFileName[MAX_PATH];       // 文件路径
    FILE *  m_pFile;                   // 文件操作指针
    UINT64  m_pos, m_len;
public:
    AVIMAINHEADER  m_avih;             // AVI主信息头
    struct strm_t
    {
        AVISTREAMHEADER      strh;
        BYTE *               strf;
        CString              strn;
        AVISUPERINDEX *      indx;
    } m_strm;
    struct ODMLExtendedAVIHeader
    {
        DWORD dwTotalFrames;
    } m_dmlh;
    AVIOLDINDEX *     m_idx1;          // 索引数据块
    UINT64        m_idx1Pos;           // 索引块在整个文件中的偏移量
public:
    CAviParser();
    ~CAviParser();
    // 设置一个待分析的AVI文件
    BOOL SetAviFile(const char * inFile);
    // 分析AVI文件的内容
    BOOL Parse(void);
private:
    UINT64 GetPos(void) { return m_pos; }
    UINT64 GetLength(void) { return m_len; }
    void Seek(UINT64 pos) { m_pos = pos; }
    void Release(void);
    HRESULT Parse(DWORD parentid, UINT64 end);
    // 从当前位置读取一定量的数据
    HRESULT Read(void* pData, LONG len);
    // 模板函数：读取一定量的数据
    template<typename T> HRESULT TRead(T& var, int offset = 0)
    {
#ifdef _DEBUG
        int size = sizeof(var);
#endif
        memset(&var, 0, sizeof(var));
        HRESULT hr = Read((BYTE*)&var + offset, sizeof(var) - offset);
        return hr;
    }
};

#endif // __H_CAviParser__

//
// CAviParser.cpp
```

```
//

#include "stdafx.h"
#include "CAviParser.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//////////////////////////////////////////////////////////////////////
CAviParser::CAviParser() : m_pos(0), m_len(0),
m_idx1(NULL), m_pFile(NULL), m_idx1Pos(0)
{
    // 初始化
    m_strm.strf = NULL;
    m_strm.indx = NULL;
}
CAviParser::~CAviParser()
{
    Release();
}
void CAviParser::Release(void)
{
    // 释放使用过的内存
    if (m_strm.strf)
    {
        delete[] m_strm.strf;
        m_strm.strf = NULL;
    }
    if (m_strm.indx)
    {
        delete[] m_strm.indx;
        m_strm.indx = NULL;
    }
    if (m_idx1)
    {
        delete[] m_idx1;
        m_idx1 = NULL;
    }
    if (m_pFile)
    {
        fclose(m_pFile);
        m_pFile = NULL;
    }
}
// 从当前位置读取一定量的数据
HRESULT CAviParser::Read(void* pData, LONG len)
{
    if (fseek(m_pFile, (long) m_pos, SEEK_SET) == 0)
    {
```

```
        fread(pData, sizeof(char), len, m_pFile);
        m_pos += len;
        return S_OK;
    }
    return E_FAIL;
}
// 设置一个待分析的AVI文件
BOOL CAviParser::SetAviFile(const char * inFile)
{
    Release();
    strcpy(mFileName, inFile);
    m_pFile = fopen(inFile, "rb");
    if (m_pFile)
    {
        fseek(m_pFile, 0, SEEK_END);
        m_len = ftell(m_pFile);
        return TRUE;
    }
    return FALSE;
}
// 分析AVI文件内容
BOOL CAviParser::Parse(void)
{
    Seek(0);
    DWORD dw[3];
    // 检查四字符码'RIFF'和'AVI ' (或'AVIX')
    if (S_OK != TRead(dw) || dw[0] != FCC('RIFF') ||
        (dw[2] != FCC('AVI ') && dw[2] != FCC('AVIX')))
    {
        return E_FAIL;
    }
    Seek(0);
    HRESULT hr = Parse(0, GetLength());
    return SUCCEEDED(hr);
}
// 这是一个递归函数，具体分析AVI文件格式
HRESULT CAviParser::Parse(DWORD parentid, UINT64 end)
{
    HRESULT hr = S_OK;
    // 一直分析到文件末尾
    while (S_OK == hr && GetPos() < end)
    {
        UINT64 pos = GetPos();

        DWORD id = 0, size;
        if (S_OK != TRead(id) || id == 0)
            return E_FAIL;
        // 如果是'RIFF'或'LIST' (列表)，则需继续分析嵌套在它们里面的数据
        if (id == FCC('RIFF') || id == FCC('LIST'))
        {
            if (S_OK != TRead(size) || S_OK != TRead(id))
                return E_FAIL;
```

```
            size += (size&1) + 8;

            TRACE(_T("CAviParser::Parse(..): LIST '%c%c%c%c'\n"),
                TCHAR((id>>0)&0xff),
                TCHAR((id>>8)&0xff),
                TCHAR((id>>16)&0xff),
                TCHAR((id>>24)&0xff));
            // 如果是'movi'列表，不用继续往里面分析
            if (id == FCC('movi'))
            {
                TRACE(_T("\nMovie LIST...\n"));
            }
            else
            {
                // 如果不是'movi'列表，则继续往嵌套的数据里面分析
                hr = Parse(id, pos + size);
            }
        }
        else
        {
            // 分析到了嵌套在'RIFF'或'LIST'里面的四字符码
            if (S_OK != TRead(size))
                return E_FAIL;

            TRACE(_T("CAviParser::Parse(..): '%c%c%c%c'\n"),
                TCHAR((id>>0)&0xff),
                TCHAR((id>>8)&0xff),
                TCHAR((id>>16)&0xff),
                TCHAR((id>>24)&0xff));

            if (parentid == FCC('INFO') && size > 0)
            {
                switch(id)
                {
                case FCC('IARL'):
                case FCC('IART'):
                case FCC('ICMS'):
                case FCC('ICMT'):
                case FCC('ICOP'):
                case FCC('ICRD'):
                case FCC('ICRP'):
                case FCC('IDIM'):
                case FCC('IDPI'):
                case FCC('IENG'):
                case FCC('IGNR'):
                case FCC('IKEY'):
                case FCC('ILGT'):
                case FCC('IMED'):
                case FCC('INAM'):
                case FCC('IPLT'):
                case FCC('IPRD'):
```

```
        case FCC('ISBJ'):
        case FCC('ISFT'):
        case FCC('ISHP'):
        case FCC('ISRC'):
        case FCC('ISRF'):
        case FCC('ITCH'):
            {
                CString str;
                if (S_OK != Read(str.GetBufferSetLength(size),
                    size))
                {
                    return E_FAIL;
                }
                break;
            }
        }
    }

    switch (id)
    {
    case FCC('avih'):   // AVI主信息头
        m_avih.fcc = FCC('avih');
        m_avih.cb  = size;
        if (S_OK != Read((BYTE*)&m_avih + 8, size))
        {
            return E_FAIL;
        }
        break;
    case FCC('strh'):   // 流的头信息
        m_strm.strh.fcc = FCC('strh');
        m_strm.strh.cb  = size;
        if (S_OK != Read((BYTE*)&m_strm.strh + 8, size))
        {
            return E_FAIL;
        }
        break;
    case FCC('strn'):   // 流的名字描述
        if (S_OK != Read(m_strm.strn.GetBufferSetLength(size),
            size))
        {
            return E_FAIL;
        }
        break;
    case FCC('strf'):   // 流的格式信息
        if (m_strm.strf)
        {
            delete[] m_strm.strf;
            m_strm.strf = NULL;
        }
        m_strm.strf = new BYTE[size];
        if (S_OK != Read(m_strm.strf, size))
        {
```

```
                    return E_FAIL;
                }
                break;
            case FCC('indx'):   // 流的索引
                if (!m_strm.indx)
                {
                    m_strm.indx = (AVISUPERINDEX*) new BYTE[size + 8];
                }
                m_strm.indx->fcc = FCC('indx');
                m_strm.indx->cb  = size;
                if (S_OK != Read((BYTE*)(AVISUPERINDEX*)m_strm.indx + 8,
                    size))
                {
                    return E_FAIL;
                }
                ASSERT(m_strm.indx->wLongsPerEntry == 4 &&
                    m_strm.indx->bIndexType == AVI_INDEX_OF_INDEXES);
                break;
            case FCC('dmlh'):
                if (S_OK != TRead(m_dmlh))
                {
                    return E_FAIL;
                }
                break;
            case FCC('vprp'):
                if (S_OK != TRead(m_vprp)) return E_FAIL;
                break;
            case FCC('idx1'):   // （可选的）AVI文件尾部的索引块！
                ASSERT(m_idx1 == NULL);
                // 将索引块数据全部读出
                m_idx1 = (AVIOLDINDEX*) new BYTE[size + 8];
                m_idx1->fcc = FCC('idx1');
                m_idx1->cb  = size;
                m_idx1Pos   = m_pos - 8;
                if (S_OK != Read((BYTE*)(AVIOLDINDEX*)m_idx1 + 8, size))
                {
                    return E_FAIL;
                }
                break;
            }
            size += (size&1) + 8;
        }
        Seek(pos + size);
    }
    return hr;
}
```

最后，再来看客户机程序这边的控制类CMediaPlayer的实现。这个类控制播放用的Filter Graph的构建和运行：首先创建一个接收网络数据的Source Filter，启动它的接收线程，但不急于将它连接到下一级的Filter；Source Filter持续接收网络数据，达到了一定的预接收量后，就可以让应用程序完成Filter Graph的构建——将Source Filter首先连接到一个Splitter

Filter，然后将Splitter Filter的所有输出Pin都连接到一个Renderer Filter（中间自动插入音视频解码Filter）；然后就可以播放媒体内容了。

```
//
// CMediaPlayer.h
//

#ifndef __H_CMediaPlayer__
#define __H_CMediaPlayer__

#include "CMediaInfo.h"
#include "CMsgReceiver.h"
#include "INetReceiver.h"

class CDXGraph;
class CDXFilter;
class CMediaPlayer : public CMsgReceiver
{
protected:
    CDXGraph *       mGraph;             // Filter Graph控制对象
    CMediaInfo       mMediaInfo;         // 指定播放的媒体节目信息
    CDXFilter *      mSourceFilter;      // 接收网络数据的Source Filter
    INetReceiver *   mINetReceiver;      // Source Filter上实现的自定义接口
    CDXFilter *      mSplitter;          // Splitter Filter的包装对象
public:
    CMediaPlayer();
    virtual ~CMediaPlayer();
    // 设置将要播放的媒体节目信息
    void SetMediaInfo(CMediaInfo & inInfo);
    // 构建/销毁Filter Graph
    BOOL Activate(void);
    void Deactive(void);
    // 完成Filter的连接
    BOOL ConnectFilters(void);
    // Filter Graph的控制，包括运行、暂停、停止
    BOOL PlayGraph(void);
    BOOL PauseGraph(void);
    BOOL StopGraph(void);
    // 网络接收Filter的各个接口函数调用
    BOOL SetStreamType(long inType);
    BOOL SetSize(long inTotalSize, long inCheckOffset);
    BOOL SetStreamSocket(SOCKET inSocket);
    BOOL StartReceiving(void);
    BOOL StopReceiving(void);
    BOOL CanConnectToDownstream(void);
    // 自定义消息处理函数
    virtual bool ReceiveMessage(MessageT inMessage, void * ioParam,
        void * ioParam2);
protected:
```

```
    BOOL CreateGraph(void);  // 创建Filter Graph对象
    void DeleteGraph(void);  // 销毁Filter Graph对象
    // 将指定Filter的所有输出Pin都连接到一个Renderer Filter
    BOOL RenderAllOutputPins(IBaseFilter * inFilter);
};

#endif // __H_CMediaPlayer__

//
// CMediaPlayer.cpp
//

#include "stdafx.h"
#include "defines.h"
#include "CMediaPlayer.h"
#include "CDXGraph.h"
#include "CDXFilter.h"

#include <initguid.h>
#include "FilterNetReceiverGuid.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////
CMediaPlayer::CMediaPlayer()
{
    // 初始化
    mGraph          = NULL;
    mSourceFilter   = NULL;
    mINetReceiver   = NULL;
    mSplitter       = NULL;
}
CMediaPlayer::~CMediaPlayer()
{
    // 释放资源
    Deactive();
}
// 设置将要播放的媒体节目信息
void CMediaPlayer::SetMediaInfo(CMediaInfo & inInfo)
{
    mMediaInfo = inInfo;
}
// --- 开始构建Filter Graph ---
// 为什么要预先创建/加入Splitter Filter，而不等到Source Filter接收到足够的数据后
// 直接使用智能连接Render一下Source Filter的输出Pin?
```

```
// 虽然直接Render Source Filter的输出Pin也能构建完整的Filter Graph，但是这个
// Render过程会尝试连接系统中所有可能的Splitter（直到找到一个能与Source Filter连
// 上的）；每次与一个Splitter试连接的过程都需要定位、读取Source Filter内的数据进行
// 格式校验，这种反反复复的过程对于Source Filter内部数据缓存的管理是个考验，同时
// Filter Graph构建的效率也是低下的。既然网络上传输的数据类型客户机已经提前知道了，
// 而且DirectShow提供了标准的MPEG1/MPEG2/AVI Splitter，为什么不预先创建并指定使
// 用它们呢？（智能连接过程是不稳定的，使用智能连接说不定还会用上一个性能很差的第三方
// Splitter，那就麻烦了！）
BOOL CMediaPlayer::Activate(void)
{
    Deactive();
    // 首先创建一个Filter Graph对象
    BOOL pass = CreateGraph();
    if (pass)
    {
        // 创建一个接收网络数据的Source Filter，然后加入到Filter Graph中
        mSourceFilter = new CDXFilter(mGraph->GetGraph(),
            CLSID_NetReceiver, "Net Source");
        pass = mSourceFilter->CreateFilter();
        // 获得Source Filter上的INetReceiver接口
        mSourceFilter->GetFilter()->QueryInterface(IID_INetReceiver,
            (void**) &mINetReceiver);
    }
    if (pass)
    {
        // 根据将要接收的网络数据类型，创建一个对应的Splitter Filter
        GUID   guid = GUID_NULL;
        switch (mMediaInfo.mFileType)
        {
        case FT_MPEG1:
        case FT_MP3:
            // DirectShow标准的MPEG-1 Stream Splitter
            guid = CLSID_MPEG1Splitter;
            break;
        case FT_MPEG2:
            // DirectShow标准的MPEG-2 Splitter
            guid = CLSID_MMSPLITTER;
            break;
        case FT_AVI:
            // DirectShow标准的AVI Splitter
            guid = CLSID_AviSplitter;
            break;
        }
        mSplitter = new CDXFilter(mGraph->GetGraph(), guid, "Splitter");
        pass = mSplitter->CreateFilter();
    }
    return pass;
}
// 连接Filter，以构建完整的Filter Graph
```

```
// 首先将Source Filter连接到Splitter Filter，然后将Splitter Filter的所有输出Pin
// 都连接到一个Renderer Filter
BOOL CMediaPlayer::ConnectFilters(void)
{
    BOOL pass = FALSE;
    if (mGraph && mSourceFilter && mSplitter)
    {
        // 获取Source Filter的输出Pin
        IPin * pOut = mSourceFilter->GetPin(FALSE);
        // 获取Splitter Filter的输入Pin
        IPin * pIn  = mSplitter->GetPin(TRUE);
        // 将Source Filter与Splitter相连接
        pass = mGraph->ConnectFilters(pOut, pIn);
        if (pass)
        {
            // 将Splitter的所有输出Pin都连接到一个Renderer Filter
            pass = RenderAllOutputPins(mSplitter->GetFilter());
        }
    }
    return pass;
}
// 释放使用过的资源
void CMediaPlayer::Deactive(void)
{
    StopGraph();
    SAFE_RELEASE(mINetReceiver);
    SAFE_DELETE(mSourceFilter);
    SAFE_DELETE(mSplitter);
    DeleteGraph();
}
// 创建Filter Graph对象
BOOL CMediaPlayer::CreateGraph(void)
{
    mGraph = new CDXGraph();
    if (mGraph->Create())
    {
        mGraph->AddMsgReceiver(this);
        return TRUE;
    }
    else
    {
        DeleteGraph();
        return FALSE;
    }
}
// 销毁Filter Graph对象
void CMediaPlayer::DeleteGraph(void)
{
    if (mGraph)
```

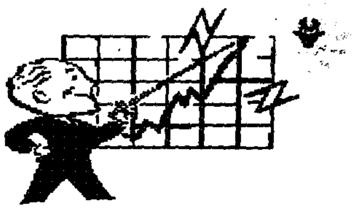
```
    {
        mGraph->RemoveMsgReceiver(this);
        delete mGraph;
        mGraph = NULL;
    }
}
// 运行Filter Graph
BOOL CMediaPlayer::PlayGraph(void)
{
    if (mGraph)
    {
        mGraph->Run();
        return TRUE;
    }
    return FALSE;
}
// 暂停Filter Graph
BOOL CMediaPlayer::PauseGraph(void)
{
    if (mGraph)
    {
        mGraph->Pause();
        return TRUE;
    }
    return FALSE;
}
// 停止Filter Graph
BOOL CMediaPlayer::StopGraph(void)
{
    if (mGraph)
    {
        mGraph->Stop();
        return TRUE;
    }
    return FALSE;
}
// 用于接收网络数据的Source Filter的各个接口函数调用
// 设置网络数据流的类型
BOOL CMediaPlayer::SetStreamType(long inType)
{
    HRESULT hr = E_FAIL;
    if (mINetReceiver)
    {
        hr = mINetReceiver->SetStreamType(inType);
    }
    return SUCCEEDED(hr);
}
// 设置网络数据流的大小，以及额外格式数据的偏移
BOOL CMediaPlayer::SetSize(long inTotalSize, long inCheckOffset)
```

```
{
    HRESULT hr = E_FAIL;
    if (mINetReceiver)
    {
        hr = mINetReceiver->SetSize(inTotalSize, inCheckOffset);
    }
    return SUCCEEDED(hr);
}
// 设置用于接收网络数据的Socket句柄
BOOL CMediaPlayer::SetStreamSocket(SOCKET inSocket)
{
    if (mINetReceiver && inSocket != INVALID_SOCKET)
    {
        mINetReceiver->SetStreamSocket(inSocket);
    }
    return TRUE;
}
// 启动Source Filter内的数据接收线程
BOOL CMediaPlayer::StartReceiving(void)
{
    HRESULT hr = E_FAIL;
    if (mINetReceiver)
    {
        hr = mINetReceiver->StartReceiving();
    }
    return SUCCEEDED(hr);
}
// 停止Source Filter内的数据接收线程
BOOL CMediaPlayer::StopReceiving(void)
{
    HRESULT hr = E_FAIL;
    if (mINetReceiver)
    {
        hr = mINetReceiver->StopReceiving();
    }
    return SUCCEEDED(hr);
}
// 询问Source Filter是否接收到了足够的数据（以让应用程序完成Filter Graph的构建）
BOOL CMediaPlayer::CanConnectToDownstream(void)
{
    HRESULT hr = S_FALSE;
    if (mINetReceiver)
    {
        hr = mINetReceiver->CanConnectToDownstream();
    }
    return (hr == S_OK);
}
// 将指定Filter的所有输出Pin都连接到一个Renderer Filter
BOOL CMediaPlayer::RenderAllOutputPins(IBaseFilter * inFilter)
```

```
{
    BOOL success = FALSE;

    if (inFilter)
    {
        IEnumPins * pinEnum = NULL;
        // 枚举Filter上所有Pin
        if (SUCCEEDED(inFilter->EnumPins(&pinEnum)))
        {
            pinEnum->Reset();

            IPin * pin = NULL;
            ULONG fetchCount = 0;
            while (SUCCEEDED(pinEnum->Next(1, &pin, &fetchCount)) &&
                fetchCount)
            {
                if (pin)
                {
                    PIN_INFO pinInfo;
                    if (SUCCEEDED(pin->QueryPinInfo(&pinInfo)))
                    {
                        pinInfo.pFilter->Release();
                        // 是否输出Pin
                        if (pinInfo.dir == PINDIR_OUTPUT)
                        {
                            // 使用智能连接
                            if (mGraph->Render(pin))
                            {
                                success = TRUE;
                            }
                        }
                    }
                    pin->Release();
                }
            }
            pinEnum->Release();
        }
    }

    return success;
}
// 自定义消息处理函数
bool CMediaPlayer::ReceiveMessage(MessageT inMessage, void * ioParam, void
    * ioParam2)
{
    // 当Filter Graph运行发生错误时将Filter Graph停止
    if (inMessage == msg_FilterGraphError)
    {
        StopGraph();
```

```
        return true;
    }
    return CMsgReceiver::ReceiveMessage(inMessage, ioParam, ioParam2);
}
```

提示：本书配套光盘的SourceCodes\Chapter03\VODDemo目录下提供了视频点播演示程序的完整实现。其中，VODServer为服务器程序，VODClient为客户机程序，FilterNetReceiver是网络接收Filter。打开WsClientServer.dsw可以同时浏览上述3个项目。



视频叠加

4.1 视频抠像

每天都看中央电视台的天气预报：一幅动态的气象云图，一位主持人站在前面“指点江山”；主持人指到哪儿，哪儿就会有天气的动态演示。你想过这是怎么做出来的吗？主持人是真的站在一幅动态的气象云图前面吗？

非也！主持人其实是在一个虚拟演播室里做节目，他的后面是一个单色的背景（背景色可能是纯蓝的），他的指指点点其实也是在“瞎”指点（他只知道要指向某个方位，但他看到的所指位置上却是一片纯色背景）。我们看到的电视效果实际上是通过计算机合成出来的——气象云图是一段视频，单一背景前的主持人动作也是一段视频，怎么将这两段视频合成呢？这里就要引入一个视频抠像的概念，即要将记录主持人动作的视频的纯色背景抠掉，只剩下主持人了，然后再叠加到气象云图的视频内容上去。

4.1.1 抠像原理

大家知道，视频是由一帧一帧连续的图像组成的。视频抠像就是要将视频中的每一幅图像中满足一定条件的那部分像素丢弃，然后把剩余部分的图像叠加到目标视频图像上去。视频抠像的典型过程如图4.1所示（假设图像A是纯蓝色背景下做运动的小孩，要将它叠加到图像B上去）。

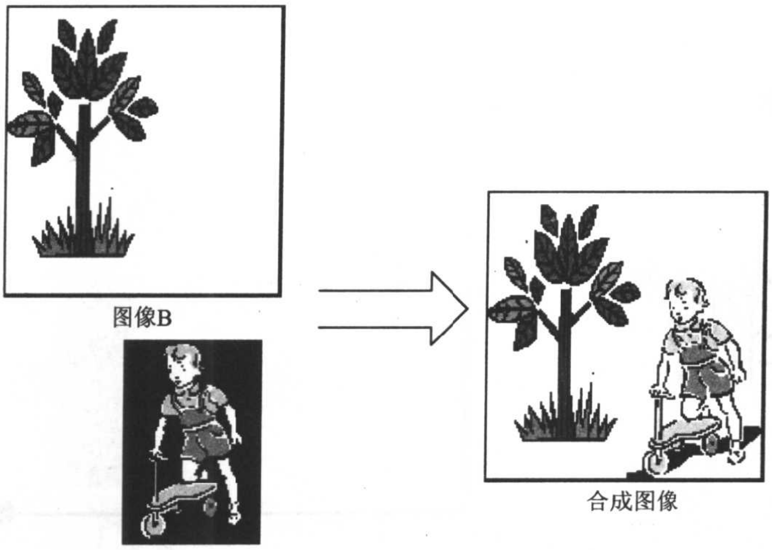


图 4.1 视频抠像叠加的过程

抠像叠加的过程为：扫描图像B，将指针一开始定位到图像A开始叠加的位置；逐一扫描图像A的像素值，如果是纯蓝色的则跳过，如果不是则取出该像素值；将从图像A中取出的像素值替换图像B中对应位置的像素值；当整幅图像A扫描完毕，抠像叠加的过程也就完成，最终生成右边的合成图像。（纯蓝色背景下做运动的小孩可能是一段视频，将视频中的每一幅图像重复上述的抠像叠加过程即可实现视频的抠像叠加。）

提示：实际应用中，如果做运动的小孩是通过摄像头实地拍摄下来的，即使小孩在纯蓝色的布景下运动，拍下来的视频图像背景也未必都是纯蓝色的。因为现实环境会存在很多干扰因素，使得理想情况下应该是纯蓝的RGB（0，0，255）背景，到最后都或多或少产生一些偏差，比如拍下来的背景像素值可能是RGB（10，0，250）或RGB（2，3，251）等。这种情况下，在进行背景色检测的时候应该考虑一定的误差！

实现视频抠像的方法有很多种，常见的有根据像素的RGB值、亮度值、Alpha值、色调等。下面介绍一种根据像素的RGB值实现抠像叠加的算法。

4.1.2 抠像算法实现

一般情况下要求叠加的两幅图像格式是一致的。为简单起见，我们还假设这两幅图像的尺寸是相同的。如果图像格式是RGB565，则理想的抠像叠加过程可以实现如下：

```
// 参数outPrimary：主视频图像数据（其他图像叠加到它上面）
// 参数inMixing：用于叠加的视频图像数据
// 背景色检测模式：DETECT_ALWAYS，每次叠加都自动检测；DETECT_ONCE，第一次叠加时检
// 测一次；CUSTOMIZE，背景色由上层调用者设置
__inline void CVideoMixController::IdealMixing16Bits(BYTE * outPrimary,
    BYTE * inMixing)
{
```

```
    // 16位的RGB格式每个像素使用2个字节表示，所以将图像数据指针转换成WORD类型的指针
    // 以方便对像素的定位
    WORD * pPrimary = (WORD *) outPrimary;
    WORD * pMixing  = (WORD *) inMixing;

    static WORD backColor;
    // 检测出背景颜色值
    WORD temp = *pMixing;
    if (mDetectType == DETECT_ALWAYS)
    {
        backColor = temp;
    }
    else if (mDetectType == DETECT_ONCE)
    {
        if (!mDetectOnceFinished)
        {
            mDetectOnceFinished = TRUE;
            backColor = temp;
        }
    }
    else
    {
        // 将RGB分量合成RGB565的格式
        backColor = (mBackColorR << 11) + (mBackColorG << 5) + mBackColorB;
    }
    // 逐一扫描图像的像素值，
    // mTotalPixels的值为输入图像的总像素数
    for (long i = 0; i < mTotalPixels; i++)
    {
        // 如果叠加图像中当前的像素值不等于背景色，则将这个像素值替换主视频图像中对应
        // 位置的像素值
        if (*pMixing != backColor)
        {
            *pPrimary = *pMixing;
        }
        // 分别指向叠加图像和主视频图像中的下一个像素
        pPrimary++;
        pMixing++;
    }
}
```

如果图像格式是RGB32，则理想的抠像叠加过程可以实现如下：

```
// 参数outPrimary：主视频图像数据（其他图像叠加到它上面）
// 参数inMixing：用于叠加的视频图像数据
__inline void CVideoMixController::IdealMixing32Bits(BYTE * outPrimary,
    BYTE * inMixing)
{
    // 32位的RGB格式每个像素使用4个字节表示，所以将图像数据指针转换成DWORD类型的指针
    // 以方便对像素的定位
    DWORD * pPrimary = (DWORD *) outPrimary;
    DWORD * pMixing  = (DWORD *) inMixing;
```

```
    static DWORD backColor;
    // 检测出背景颜色值
    DWORD temp = *pMixing;
    if (mDetectType == DETECT_ALWAYS)
    {
        backColor = temp;
    }
    else if (mDetectType == DETECT_ONCE)
    {
        if (!mDetectOnceFinished)
        {
            mDetectOnceFinished = TRUE;
            backColor = temp;
        }
    }
    else
    {
        // 将RGB分量合成RGB32的格式
        backColor = (mBackColorR << 16) + (mBackColorG << 8) + mBackColorB;
    }
    // 逐一扫描图像的像素值，mTotalPixels的值为输入图像的总像素数
    for (long i = 0; i < mTotalPixels; i++)
    {
        // 如果叠加图像中当前的像素值不等于背景色，则将这个像素值替换主视频图像中对应
        // 位置的像素值
        if (*pMixing != backColor)
        {
            *pPrimary = *pMixing;
        }
        // 分别指向叠加图像和主视频图像中的下一个像素
        pPrimary++;
        pMixing++;
    }
}
```

如果主视频图像要与实地拍摄的纯色背景下的视频叠加，那又该如何实现呢？与理想情况下的抠像叠加相比，这时候重要的是要考虑背景色的偏差。如果图像格式是RGB565，则这个抠像叠加过程可以实现如下：

```
// 参数outPrimary：主视频图像数据（其他图像叠加到它上面）
// 参数inMixing：用于叠加的视频图像数据
// 变量mBackDrift定义背景色分量允许的最大偏移值
// #define RGB565_MASK_RED    0xF800
// #define RGB565_MASK_GREEN  0x07E0
// #define RGB565_MASK_BLUE   0x001F
// #define DEFAULT_RGB565_DRIFT   8  // 默认的RGB分量允许的最大偏移值
__inline void CVideoMixController::ActualMixing16Bits(BYTE * outPrimary,
    BYTE * inMixing)
{
    WORD * pPrimary = (WORD *) outPrimary;
    WORD * pMixing  = (WORD *) inMixing;
    static BYTE backRed, backGreen, backBlue;
```

```
    static BYTE mixingR, mixingG, mixingB;
    // 检测出背景颜色值，并且分离出RGB各分量
    WORD temp = *pMixing;
    if (mDetectType == DETECT_ALWAYS)
    {
        backRed     = (temp & RGB565_MASK_RED) >> 11;
        backGreen   = (temp & RGB565_MASK_GREEN) >> 5;
        backBlue    = temp & RGB565_MASK_BLUE;
    }
    else if (mDetectType == DETECT_ONCE)
    {
        if (!mDetectOnceFinished)
        {
            mDetectOnceFinished = TRUE;
            backRed     = (temp & RGB565_MASK_RED) >> 11;
            backGreen   = (temp & RGB565_MASK_GREEN) >> 5;
            backBlue    = temp & RGB565_MASK_BLUE;
        }
    }
    else
    {
        backRed     = mBackColorR;
        backGreen   = mBackColorG;
        backBlue    = mBackColorB;
    }
    // 逐一扫描图像的像素值，mTotalPixels的值为输入图像的总像素数
    for (long i = 0; i < mTotalPixels; i++)
    {
        temp        = *pMixing;
        mixingR     = (temp & RGB565_MASK_RED) >> 11;
        mixingG     = (temp & RGB565_MASK_GREEN) >> 5;
        mixingB     = temp & RGB565_MASK_BLUE;
        // 如果叠加图像中当前像素值的各分量值都没有超过允许的最大偏移值，则认为当前像
        // 素是背景色，将其跳过
        if (MyAbs(mixingR - backRed) < mBackDrift &&
            MyAbs(mixingG - backGreen) < mBackDrift &&
            MyAbs(mixingB - backBlue) < mBackDrift)
        {
            // 指向主视频图像的下一个像素
            pPrimary++;
        }
        else
        {
            // 将叠加图像中当前像素值替换主视频图像中对应位置的像素值
            *pPrimary = *pMixing;
            // 指向主视频图像的下一个像素
            pPrimary++;
        }
        // 指向叠加图像的下一个像素
        pMixing++;
    }
}
```

```
// 自定义的取绝对值的函数
__inline BYTE CVideoMixController::MyAbs(__int16 inValue)
{
    if (inValue < 0)
    {
        return -inValue;
    }
    return inValue;
}
```

如果图像格式是RGB32，则现实环境下的抠像叠加过程可以实现如下：

```
// 参数outPrimary：主视频图像数据（其他图像叠加到它上面）
// 参数inMixing：用于叠加的视频图像数据
// #define RGB32_MASK_RED    0x00FF0000
// #define RGB32_MASK_GREEN  0x0000FF00
// #define RGB32_MASK_BLUE   0x000000FF
// #define DEFAULT_RGB32_DRIFT    24  // 默认的RGB分量允许的最大偏移值
__inline void CVideoMixController::ActualMixing32Bits(BYTE * outPrimary,
    BYTE * inMixing)
{
    DWORD * pPrimary = (DWORD *) outPrimary;
    DWORD * pMixing  = (DWORD *) inMixing;
    static BYTE backRed, backGreen, backBlue;
    static BYTE mixingR, mixingG, mixingB;
    // 检测出背景颜色值，并且分离出RGB各分量
    DWORD temp = *pMixing;
    if (mDetectType == DETECT_ALWAYS)
    {
        backRed    = BYTE((temp & RGB32_MASK_RED) >> 16);
        backGreen  = BYTE((temp & RGB32_MASK_GREEN) >> 8);
        backBlue   = BYTE(temp & RGB32_MASK_BLUE);
    }
    else if (mDetectType == DETECT_ONCE)
    {
        if (!mDetectOnceFinished)
        {
            mDetectOnceFinished = TRUE;
            backRed    = BYTE((temp & RGB32_MASK_RED) >> 16);
            backGreen  = BYTE((temp & RGB32_MASK_GREEN) >> 8);
            backBlue   = BYTE(temp & RGB32_MASK_BLUE);
        }
    }
    else
    {
        backRed   = mBackColorR;
        backGreen = mBackColorG;
        backBlue  = mBackColorB;
    }
    // 逐一扫描图像的像素值，mTotalPixels的值为输入图像的总像素数
    for (long i = 0; i < mTotalPixels; i++)
    {
        temp    = *pMixing;
```

```
            mixingR = BYTE((temp & RGB32_MASK_RED) >> 16);
            mixingG = BYTE((temp & RGB32_MASK_GREEN) >> 8);
            mixingB = BYTE(temp & RGB32_MASK_BLUE);
            // 如果叠加图像中当前像素值的各分量值都没有超过允许的最大偏移值，则认为当前像
            // 素是背景色，将其跳过
            if (MyAbs(mixingR - backRed) < mBackDrift &&
                MyAbs(mixingG - backGreen) < mBackDrift &&
                MyAbs(mixingB - backBlue) < mBackDrift)
            {
                // 指向主视频图像的下一个像素
                pPrimary++;
            }
            else
            {
                // 将叠加图像中当前像素值替换主视频图像中对应位置的像素值
                *pPrimary = *pMixing;
                // 指向主视频图像的下一个像素
                pPrimary++;
            }
            // 指向叠加图像的下一个像素
            pMixing++;
        }
    }
```

上述4种抠像算法实现的效率对比如表4.1所示（测试环境为Windows 2000操作系统，安装了DirectX 9.0，P4 1.6GHz的CPU，256M内存，80G硬盘，测试图像大小为720 x 480）。

表4.1 4种抠像算法实现的效率对比

算法	执行一次抠像叠加需要的时间
理想情况下的RGB565抠像叠加	1ms
理想情况下的RGB32抠像叠加	1ms~2ms
现实环境下的RGB565抠像叠加	5ms~6ms
现实环境下的RGB32抠像叠加	5ms~6ms

提示：上述4种算法实现函数都被声明为内联函数（__inline），以提高函数调用的效率。另外，编译时使用了VC的Maximize Speed优化，这一点很重要。以现实环境下的RGB32抠像叠加为例，如果不使用VC的优化设置，执行一次抠像叠加需要的时间将是20ms（几乎是采用VC优化设置算法的4倍）。VC的上述优化设置方法如下：将当前的编译环境设置为Win32 Release，执行VC的菜单命令Project | Settings，在随后弹出的对话框中单击C/C++标签，然后在Category下拉列表中选择Optimizations，随后在Optimizations下拉列表中选择Maximize Speed，在Inline function expansion下拉列表中选择Any Suitable即可。

4.1.3 MMX算法优化

多媒体数据的处理，效率是很重要的。如果VC的优化措施仍然不能起到很好的效果，那就要考虑自己来做优化了。处理大块大块的图像数据，我们最先想到的是使用MMX指令。以理想情况下的RGB565格式的抠像叠加为例，下面是使用MMX指令的实现方法：

```
// 参数outPrimary：主视频图像数据（其他图像叠加到它上面）
// 参数inMixing：用于叠加的视频图像数据
__inline void CVideoMixController::IdealMMXMixing16Bits(BYTE * outPrimary,
    BYTE * inMixing)
{
    static int backR    = mBackColorR;  // 背景色的R分量
    static int backG    = mBackColorG;  // 背景色的G分量
    static int backB    = mBackColorB;  // 背景色的B分量
    static __int64 mask = 0xFFFFFFFFFFFFFFFF;
    static long  pixelCount = mTotalPixels;  // 图像的总像素数

    __asm
    {
        mov esi, inMixing    ; 叠加图像数据
        mov edi, outPrimary  ; 主视频图像数据
        ; MMX指令每次能处理64位数据，也就是一次处理4个像素
        mov ecx, pixelCount
        shr ecx, 2           ; 计算循环次数

        mov eax, backR  ; 将RGB各分量组成RGB565的16位形式
        mov ebx, backG
        shl eax, 11
        shl ebx, 5
        add eax, ebx
        add eax, backB

        pxor mm6, mm6   ; 清零
        movd mm6, eax   ; 将背景色打包成4像素的形式
        punpcklwd mm6, mm6
        punpckldq mm6, mm6
        movq mm7, mask

next_4pixel:
        movq mm0, [esi]      ; 取出叠加图像中的4个像素
        movq mm1, [edi]      ; 取出主视频图像中的4个像素
        movq mm2, mm6

        pcmpeqw mm2, mm0 ; 检测叠加图像中的背景色
        pand mm1, mm2    ; 将叠加图像中背景色像素对应位置的主视频图像的像素保留
        pxor mm2, mm7    ; 得到叠加图像中的非背景色的像素屏蔽信息
        pand mm0, mm2    ; 提取叠加图像中的非背景色像素
        paddw mm1, mm0   ; 将叠加图像中的非背景色像素叠加到主视频图像的对应位置
        movq [edi], mm1  ; 将完成叠加的4个像素保存

        add esi, 8  ; 指向叠加图像中的下4个像素
```

```
            add edi, 8  ; 指向主视频图像中的下4个像素
            sub ecx, 1  ; 循环计数减1
            cmp ecx, 0
            jg next_4pixel   ; 跳回去处理下4个像素

            emms  ; 清除MMX状态
        }
    }
```

另外，在使用上述MMX指令实现算法之前，最好先检测一下你的CPU是否支持MMX指令集，方法如下：

```
// 判断当前的CPU是否支持MMX指令集，函数返回TRUE表示支持，返回FALSE表示不支持
BOOL CVideoMixController::IsMMXSupported(void)
{
    BOOL  isSupported = TRUE;
    __asm
    {
        mov eax, 1
        CPUID
        test edx, 00800000H
        jnz found
        mov isSupported, 0
found:
    }
    return isSupported;
}
```

提示：上述使用MMX指令实现的算法效率并不高，执行一次抠像叠加需要的时间平均为2ms（还不如VC自带的优化措施）。这里要引入的只是一个多媒体数据处理经常要考虑的“优化”问题。有兴趣的读者可以将上述算法实现进一步优化。

小知识：MMX 指令集

MMX指令集（Multi-Media eXtension）实质是一种SIMD（单指令多数据）数据处理方式，它由Intel公司开发，主要用于增强CPU对密集型数据的计算能力。MMX指令集共有57条多媒体指令，使用8个64位的寄存器（实际上就是浮点寄存器），分别被命名为mm0、mm1…mm7。这些寄存器每个都可以被看成是8个BYTE或4个WORD或2个DWORD或1个QWORD。下面就来介绍一些常用的MMX指令。（在下面的介绍中，使用[]来区分不同数据类型的指令，如mov[d,q]实际上是两个指令：movd和movq，前者是DWORD类型数据的转移指令，后者是QWORD类型数据的转移指令。）

1. 数据转移指令

mov[d,q]，一次性将32位或64位的数据在寄存器与内存之间转移。操作DWORD时，MMX寄存器只使用低32位。

2. 算术指令

padd[b,w,d]，无符号数加。

padds[b,w]，饱和模式的有符号数加。

paddus[b,w]，饱和模式的无符号数加。

psub[b,w,d]，无符号数减。

psubs[b,w]，饱和模式的有符号数减。

psubus[b,w]，饱和模式的无符号数减。

pmulhw，WORD类型的有符号数相乘后取高16位。

pmullw，WORD类型的有符号数相乘后取低16位。

pmaddwd，WORD类型的向量点乘运算（生成DWORD）。如4个WORD为A1A2A3A4，另外4个WORD为B1B2B3B4，使用这条指令操作这两个数的运算过程为：（A1*B1＋A2*B2）和（A3*B3＋A4*B4）。

需要说明的是，上述饱和模式的加/减指令很有用，特别是在进行图像处理的时候（如调整图像的亮度等），经常会碰到运算后的结果饱和溢出的问题，这时使用这些饱和指令就很方便。饱和运算指令能做到，当运算结果大于一个上限或小于一个下限的时候，结果仍然等于上限或下限。对于一个BYTE，有符号数的范围为-128~127，无符号数的范围为0~255。另外关于pmulhw和pmullw两个指令，将两个WORD相乘以后将得到一个DWORD，但这两个指令一次只返回高16位或低16位；完整的运算结果需要组合执行pmulhw和pmullw，也就是说要进行两次乘法运算。

3. 移位指令

psll[w,d,q]，逻辑左移。

psrl[w,d,q]，逻辑右移。

psra[w,d,q]，算术右移。

4. 逻辑指令

pand，位与操作。

pandn，将目的操作数取反后再和源操作数做位与操作。如mm0的值为0xFFFF0000FFFF0000，mm1的值为0x0000000012345678，执行pandn mm0，mm1之后mm0的值将变为0x0000000000005678。（计算过程：0xFFFF0000FFFF0000按位取反得到0x0000FFFF0000FFFF，再将0x0000FFFF0000FFFF与0x0000000012345678进行位与操作得到0x0000000000005678。）

por，位或操作。

pxor，按位异或操作。

5. 比较指令

pcmpeq[b,w,d]，判断目的操作数与源操作数是否相等。如果相等，所有位都置为1；否则置为0。如mm0的值为0xF1F2F3F480818283，mm1的值为0xF100FFF400000083，执行pcmpeqb mm0, mm1之后mm0的值将变为0xFF0000FF000000FF。

pcmpgt[b,w,d]，判断有符号目的操作数是否大于源操作数。如果大于，所有位都置为1；否则置为0。如mm0的值为0x7172737480718283，mm1的值为0x71007F8400008088，执行pcmpgtb mm0, mm1之后mm0的值将变为0x00FF00FF00FFFF00。

值得注意的是，上述两个比较指令在字符串的快速比较、视频的色键抠像叠加等方面有较多的用武之地。

6. 数据打包/拆包指令

packuswb，有符号的WORD带饱和打包成无符号的BYTE，结果将目的操作数生成的BYTE放在低位，源操作数生成的BYTE放在高位。如mm0的值为0x2000010000880065，mm1的值为0x3000040000890066，执行packuswb mm0, mm1之后mm0的值将变为0xFFFF8966FFFF8865。也就是说，mm0中的0x0065打包成0x65放在结果的第1个字节（从低往高数），0x0088打包成0x88放在结果的第2个字节，0x0100饱和了则打包成0xFF放在结果的第3个字节，0x2000饱和了则打包成0xFF放在结果的第4个字节。mm1中的各个WORD类似处理，生成的BYTE依次放在结果的高地址4个字节。

packss[wb,dw]，有符号数带饱和打包成有符号数（从WORD到BYTE或从DWORD到WORD）。与packuswb类似。

punpackh[bw,wd,dq]，将两个操作数的高位[BYTE或WORD或DWORD]交错打包成[WORD或DWORD或QWORD]（交错打包时源操作数放在高位，目的操作数放在低位）。如mm0的值为0xA1A2A3A4A5A6A7A8，mm1的值为0xB1B2B3B4B5B6B7B8，执行punpckhbw mm0, mm1之后mm0的值将变为0xB1A1B2A2B3A3B4A4。

punpackl[bw,wd,dq]，将两个操作数的低位[BYTE或WORD或DWORD]交错打包成[WORD或DWORD或QWORD]。与punpackh[bw,wd,dq]类似。如mm6的值为0x0000000000005678，执行punpcklwd mm6, mm6之后mm6的值将变为0x0000000056785678，再执行punpckldq mm6, mm6之后mm6的值将变为0x5678567856785678。

7. MMX状态清除指令

emms，清除MMX状态。

值得注意的是，MMX指令的执行实际借助于浮点寄存器。因此，MMX指令的使用切忌与浮点运算混合进行。在所有的MMX指令执行完之后，还要记得执行一条emms指令，以清除MMX状态（将浮点寄存器的使用权交出）。另外，MMX指令集对CPU的浮点运算能力没有什么贡献，因此MMX指令集在3D制作上没有实际意义（3D运算多为浮点运算）。

4.1.4 开发抠像Filter

既然抠像叠加的算法已经实现了，现在写一个Filter应该不是很难了。（抠像叠加的算法实现是“内核”，Filter只是一个应用框架而已！）这个Filter的名字暂定为HQ Video Keyer，应该有两个输入Pin和一个输出Pin。由于该Filter对输入Sample处理后数据不会扩大，输入输出的媒体类型也没有改变，所以Filter的父类可以选择CTransInPlaceFilter。只是CTransInPlaceFilter在基类库中实现的是一进一出的Filter，在这里能用吗？此外，两个输入Pin上接收到的Sample如何同步？怎么驱动抠像叠加功能的完成？这些都是亟待解决的问题。

如图4.2所示，CTransInPlaceFilter类已经实现的一对输入/输出Pin名字为Input和Output，

让主视频流从这个Input Pin进入，然后直接在这个Pin的Receive函数中调用控制对象进行抠像叠加，之后通过Output Pin将完成叠加的数据发送下去。也就是说，由Input Pin来驱动抠像叠加功能的完成。另外，Mixing In是一个自定义的输入Pin，用于接收叠加视频流数据。这个Mixing In Pin收到Sample数据之后，直接将数据交给控制对象（控制对象会对数据进行缓存以备后用）。这样，Input Pin与Mixing In Pin之间就通过控制对象实现了同步。

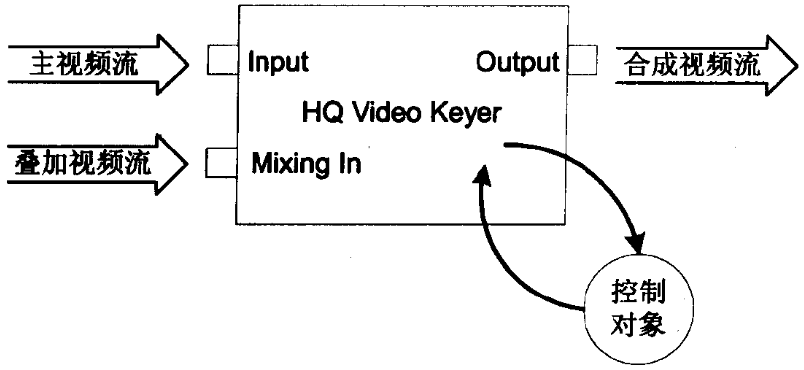


图 4.2　HQ Video Keyer Filter 的功能实现

下面，首先来看Mixing In Pin的实现。实现这个Pin的类为CBranchInputPin，它从CBaseInputPin类派生。这个Pin上实现的函数（如Receive、EndOfStream、BeginFlush、EndFlush等）大都“委托”给了Filter上的控制对象。

```
//
// CBranchInputPin.h
//

#ifndef __H_CBranchInputPin__
#define __H_CBranchInputPin__

class CFilterVideoKeyer;
class CBranchInputPin : public CBaseInputPin
{
private:
    CFilterVideoKeyer *     mFilter;        // Filter对象指针
    CCritSec                mSyncStream;    // 数据流同步
public:
    CBranchInputPin(CFilterVideoKeyer * inFilter, HRESULT * phr, LPCWSTR
        inPinName);
    ~CBranchInputPin();
    // 在Pin的连接过程中检查媒体类型
    HRESULT CheckMediaType(const CMediaType* mtIn);
    STDMETHODIMP Receive(IMediaSample * pSample);
    STDMETHODIMP EndOfStream(void);
    STDMETHODIMP BeginFlush(void);
    STDMETHODIMP EndFlush(void);
public:
    CMediaType& CurrentMediaType(void) { return m_mt; };
};
```

```
#endif // __H_CBranchInputPin__

//
// CBranchInputPin.cpp
//

#include <streams.h>
#include "FilterVideoKeyer.h"
#include "CBranchInputPin.h"
/////////////////////////////////////////////////////////////////////////////
CBranchInputPin::CBranchInputPin(CFilterVideoKeyer * inFilter, HRESULT *
    phr, LPCWSTR inPinName) :
CBaseInputPin(NAME("Branch In"), inFilter, &inFilter->m_csFilter, phr,
    inPinName)
{
    mFilter = inFilter;
}
CBranchInputPin::~CBranchInputPin()
{
}
// 在Pin的连接过程中检查媒体类型，必须保证这个Pin上使用的媒体类型与主视频流输入Pin上
// 的媒体类型一致
HRESULT CBranchInputPin::CheckMediaType(const CMediaType* mtIn)
{
    // 要求主视频流输入Pin首先连接
    if (mFilter->m_pInput && mFilter->m_pInput->IsConnected())
    {
        CMediaType mtMain = mFilter->m_pInput->CurrentMediaType();
        if (mtMain.majortype == mtIn->majortype &&
            mtMain.subtype == mtIn->subtype &&
            mtMain.formattype == mtIn->formattype)
        {
            // 检查输入图像的宽、高是否相同
            VIDEOINFOHEADER * pMain = (VIDEOINFOHEADER *) mtMain.Format();
            VIDEOINFOHEADER * pMix  = (VIDEOINFOHEADER *) mtIn->Format();
            if (pMain->bmiHeader.biWidth == pMix->bmiHeader.biWidth &&
                pMain->bmiHeader.biHeight == pMix->bmiHeader.biHeight)
            {
                return NOERROR;
            }
        }
    }
    return E_FAIL;
}
// 将该Pin上收到的Sample数据直接交给Filter上的控制对象
STDMETHODIMP CBranchInputPin::Receive(IMediaSample * pSample)
{
    CAutoLock lck(&mSyncStream);
    ASSERT(pSample);
    HRESULT hr = CBaseInputPin::Receive(pSample);
    if (S_OK == hr)
    {
```

```
            hr = mFilter->mController.ReceiveMixingVideo(pSample);
        }
        return hr;
    }
    // 将叠加视频流结束的消息通知Filter上的控制对象
    STDMETHODIMP CBranchInputPin::EndOfStream(void)
    {
        CAutoLock lck(&mSyncStream);
        HRESULT hr = CheckStreaming();
        if (S_OK == hr)
        {
            hr = mFilter->mController.MixingEndOfStream();
        }
        return hr;
    }
    // 将BeginFlush事件告诉Filter上的控制对象
    STDMETHODIMP CBranchInputPin::BeginFlush(void)
    {
        CAutoLock  lck(&mFilter->m_csFilter);
        if (!IsConnected())
        {
            return VFW_E_NOT_CONNECTED;
        }
        HRESULT hr = CBaseInputPin::BeginFlush();
        if (FAILED(hr))
        {
            return hr;
        }
        return mFilter->mController.MixingBeginFlush();
    }
    // 将EndFlush事件告诉Filter上的控制对象
    STDMETHODIMP CBranchInputPin::EndFlush(void)
    {
        CAutoLock lck(&mFilter->m_csFilter);
        if (!IsConnected())
        {
            return VFW_E_NOT_CONNECTED;
        }
        HRESULT hr = mFilter->mController.MixingEndFlush();
        if (FAILED(hr))
        {
            return hr;
        }
        return CBaseInputPin::EndFlush();
    }
```

然而，CTransInPlaceFilter类实现的是一进一出的Filter框架，如何将上述的Mixing In Pin增加进去呢？下面来看Filter类的框架函数实现，重点看GetPinCount、GetPin、FindPin 3个函数（它们将为CTransInPlaceFilter类扩展一个新的Mixing In Pin）。

```
    //
    // CFilterVideoKeyer.h
```

```
//

#ifndef __H_CFilterVideoKeyer__
#define __H_CFilterVideoKeyer__

#include "CVideoMixController.h"

class CBranchInputPin;
class CFilterVideoKeyer : public CTransInPlaceFilter
{
private:
    CBranchInputPin *     mMixingInput; // 新增的Mixing In Pin
    CVideoMixController   mController;  // 控制对象

    // 其他成员（省略）
    // ……
};
#endif // __H_CFilterVideoKeyer__

//
// CFilterVideoKeyer.cpp
//
// 在Filter类的构造函数中创建Mixing In Pin
CFilterVideoKeyer::CFilterVideoKeyer(TCHAR *tszName, LPUNKNOWN punk,
    HRESULT *phr) :
CTransInPlaceFilter(tszName, punk, CLSID_VideoKeyer, phr)
{
    mMixingInput = new CBranchInputPin(this, phr, L"Mixing In");
}
// 在Filter类的析构造函数中销毁Mixing In Pin
CFilterVideoKeyer::~CFilterVideoKeyer()
{
    delete mMixingInput;
}
// 得到该Filter拥有的Pin的个数
int CFilterVideoKeyer::GetPinCount()
{
    return 3;  // 该Filter有3个Pin
}
// 根据索引，分别得到该Filter上的各个Pin
CBasePin * CFilterVideoKeyer::GetPin(int n)
{
    switch (n)
    {
    case 0:
        return CTransInPlaceFilter::GetPin(0);
    case 1:
        return CTransInPlaceFilter::GetPin(1);
    case 2:
        return mMixingInput;
    }
    return NULL;
```

```
}
// 根据ID来寻找Pin
STDMETHODIMP CFilterVideoKeyer::FindPin(LPCWSTR Id, IPin ** ppPin)
{
    CheckPointer(ppPin,E_POINTER);
    ValidateReadWritePtr(ppPin,sizeof(IPin *));
    if (0 == lstrcmpW(Id, L"In"))
    {
        *ppPin = GetPin(0);
    }
    else if (0 == lstrcmpW(Id, L"Out"))
    {
        *ppPin = GetPin(1);
    }
    else if (0 == lstrcmpW(Id, L"Mixing In"))
    {
        *ppPin = GetPin(2);
    }
    else
    {
        *ppPin = NULL;
        return VFW_E_NOT_FOUND;
    }

    HRESULT hr = NOERROR;
    if (*ppPin)
    {
        (*ppPin)->AddRef();  // 增加一个引用计数
    }
    else
    {
        hr = E_OUTOFMEMORY;
    }
    return hr;
}
// Input Pin在连接时进行的媒体类型检查
HRESULT CFilterVideoKeyer::CheckInputType(const CMediaType* mtIn)
{
    // 不支持媒体类型的动态改变
    if (!IsStopped())
    {
        return E_FAIL;
    }
    // 检查是不是支持的RGB格式
    if (!IsSupportedRGB(mtIn))
    {
        return E_FAIL;
    }
    return NOERROR;
}
// 检查输入的媒体类型是不是RGB565或者RGB32
BOOL CFilterVideoKeyer::IsSupportedRGB(const CMediaType* mtIn)
```

```
{
    if (mtIn->majortype == MEDIATYPE_Video &&
        mtIn->formattype == FORMAT_VideoInfo)
    {
        if (mtIn->subtype == MEDIASUBTYPE_RGB565 ||
            mtIn->subtype == MEDIASUBTYPE_RGB32)
        {
            return TRUE;
        }
    }
    return FALSE;
}
// Input Pin完成连接之后，将输入图像的格式信息设置给控制对象
HRESULT CFilterVideoKeyer::CompleteConnect(PIN_DIRECTION direction,IPin
    *pReceivePin)
{
    HRESULT hr = CTransInPlaceFilter::CompleteConnect(direction,
        pReceivePin);
    if (SUCCEEDED(hr) && direction == PINDIR_INPUT)
    {
        CMediaType mt = m_pInput->CurrentMediaType();
        mController.SetInputVideoInfo((VIDEOINFOHEADER *)mt.Format());
    }
    return NOERROR;
}
// Input Pin上收到主视频图像后，调用控制对象进行抠像叠加
HRESULT CFilterVideoKeyer::Transform(IMediaSample *pSample)
{
    if (mMixingInput->IsConnected())
    {
        mController.DoMixing(pSample);
    }
    return NOERROR;
}
// 当主视频流结束时通知控制对象
HRESULT CFilterVideoKeyer::EndOfStream(void)
{
    mController.MainVideoEndOfStream();
    return CTransInPlaceFilter::EndOfStream();
}
// Filter Graph开始运行时进行初始化
HRESULT CFilterVideoKeyer::StartStreaming()
{
    mController.Start();
    return NOERROR;
}
// Filter Graph进入停止状态时的处理
STDMETHODIMP CFilterVideoKeyer::Stop()
{
    CAutoLock lck1(&m_csFilter);
    if(m_State == State_Stopped) {
        return NOERROR;
```

```
    }
    ASSERT(m_pInput == NULL || m_pOutput != NULL);
    if(m_pInput == NULL || m_pInput->IsConnected() == FALSE ||
        m_pOutput->IsConnected() == FALSE) {
        m_State = State_Stopped;
        m_bEOSDelivered = FALSE;
        return NOERROR;
    }

    ASSERT(m_pInput);
    ASSERT(m_pOutput);
    // 注意这里，让控制对象停止叠加工作
    mController.Stop();
    mMixingInput->Inactive();
    m_pInput->Inactive();

    CAutoLock lck2(&m_csReceive);
    m_pOutput->Inactive();
    HRESULT hr = StopStreaming();
    if(SUCCEEDED(hr)) {
        m_State = State_Stopped;
        m_bEOSDelivered = FALSE;
    }
    return hr;
}
```

到这里，Filter框架基本实现好了。接下去就要看控制类的实现了。因为控制类要对Mixing In Pin进来的数据进行缓存，因此它使用了一个自定义类CFrameList。这个类分配一定数量的内存空间，用于缓存叠加图像帧；当缓存已满，则缓存中最早的一帧将被丢弃，以留出空间来保存新进来的图像帧。

```
//
// CFrameList.h
//

#ifndef __H_CFrameList__
#define __H_CFrameList__

class CFrameList
{
private:
    long        mFrameSize;            // 图像帧的大小（以字节为单位）
    long        mMaxFrameCount;        // 缓存中最多可以容纳的图像帧数
    long        mLastFrameOffset;      // 最后一帧存储的位置
    long        mFrameCounter;         // 帧计数器
    CCritSec    mSyncBuffer;           // 缓存读写同步
    BYTE *      mFrameBuffer;          // 缓存地址
    BOOL        mReachEnd;             // 指示调用者已将所有帧都送进来了
    long        mIndexWhenEndReached;  // 用于缓存中残留帧的读出
private:
    // 推进一帧的位置（必要时将最早的一帧丢弃）
    __inline void MoveOneFrameUp(void);
```

```
public:
    CFrameList(long inFrameSize, long inMaxCount);
    ~CFrameList();
    // 增加一帧进入缓存
    void AddFrame(BYTE * inFrame);
    // 从缓存中读出一帧
    void GetFrame(PBYTE * outFrame);
    // 从缓存中读数据完成后调用该函数解锁同步对象
    void ReleaseFrame(void);
    // 判断缓存是否已满
    BOOL IsFrameBufferFull(void);
    // 设置调用者是否已经将所有帧都送进来了
    void SetReachEnd(BOOL inIsEnd);
    // 清空缓存
    void Flush(void);
};
#endif // __H_CFrameList__

//
// CFrameList.cpp
//

#include <streams.h>
#include "CFrameList.h"
///////////////////////////////////////////////////////////////////////////
CFrameList::CFrameList(long inFrameSize, long inMaxCount)
{
    // 参数初始化，申请缓存空间
    mFrameCounter       = 0;
    mFrameSize          = inFrameSize;
    mMaxFrameCount      = inMaxCount;
    // 指向缓存中保存最后一帧的地址
    mLastFrameOffset    = (mMaxFrameCount - 1) * mFrameSize;
    mFrameBuffer        = new BYTE[inFrameSize * inMaxCount];
    mReachEnd           = FALSE;
    mIndexWhenEndReached = 0;
}
CFrameList::~CFrameList()
{
    // 释放缓存空间
    if (mFrameBuffer)
    {
        delete[] mFrameBuffer;
        mFrameBuffer = NULL;
    }
}
// 推进一帧的位置（必要时将最早的一帧丢弃）
__inline void CFrameList::MoveOneFrameUp(void)
{
    for (int i = 1; i < mMaxFrameCount; i++)
    {
        BYTE * pCurrent = mFrameBuffer + i * mFrameSize;
```

```
        memcpy(pCurrent - mFrameSize, pCurrent, mFrameSize);
    }
}
// 增加一帧进入缓存
void CFrameList::AddFrame(BYTE * inFrame)
{
    CAutoLock  lck(&mSyncBuffer);
    // 将缓存中已有的图像帧推前一帧的位置
    MoveOneFrameUp();
    // 在缓存尾部加入新进来的这一帧
    memcpy(mFrameBuffer + mLastFrameOffset, inFrame, mFrameSize);
    mFrameCounter++;  // 帧计数
}
// 从缓存中读出一帧
void CFrameList::GetFrame(PBYTE * outFrame)
{
    mSyncBuffer.Lock();  // 锁定同步对象
    // 如果缓存的图像帧没有填满缓存，暂时不让读出
    if (mFrameCounter < mMaxFrameCount)
    {
        *outFrame = NULL;
    }
    else
    {
        // 让缓存头上的一帧图像（最早进来的图像）被读出
        *outFrame = mFrameBuffer;
        // 如果外部不再有新的图像帧进来，则必须将残留的图像帧读出
        if (mReachEnd)
        {
            if (mIndexWhenEndReached < mMaxFrameCount)
            {
                *outFrame = mFrameBuffer + mIndexWhenEndReached *
                    mFrameSize;
                mIndexWhenEndReached++;
            }
            else
            {
                *outFrame = NULL;
            }
        }
    }
    // 解锁同步对象
    if (*outFrame == NULL)
    {
        mSyncBuffer.Unlock();
    }
}
// 当一帧图像数据读出后需要调用该函数进行同步对象解锁
void CFrameList::ReleaseFrame(void)
{
    mSyncBuffer.Unlock();
}
```

```
// 判断缓存是否已满
BOOL CFrameList::IsFrameBufferFull(void)
{
    return (mFrameCounter >= mMaxFrameCount);
}
// 清空缓存
void CFrameList::Flush(void)
{
    CAutoLock lck(&mSyncBuffer);
    mFrameCounter = 0;
}
// 设置调用者是否已经将所有帧都送进来了
void CFrameList::SetReachEnd(BOOL inIsEnd)
{
    mReachEnd = inIsEnd;
    mIndexWhenEndReached = 0;
}
```

控制类CVideoMixController的实现如下：

```
//
// CVideoMixController.h
//

#ifndef __H_CVideoMixController__
#define __H_CVideoMixController__

#define BUFFERED_FRAME_COUNT   3

#include "VideoKeyerDefs.h"
#include "CFrameList.h"
#include "IVideoKeyer.h"

class CVideoMixController
{
private:
    CFrameList *     mMixingFrames;           // 用于缓存叠加视频图像
    long             mMixingFrameSize;        // 图像帧大小
    long             mTotalPixels;            // 图像的总像素个数
    WORD             mBitCount;               // 每个像素占用的位数
    BOOL             mEOSMixing;              // 指示叠加视频流结束
    BOOL             mEOSMainVideo;           // 指示主视频流结束
    BOOL             mStopped;                // 指示要求叠加工作结束
    BOOL             mMixingFlushing;         // 叠加视频流Flushing的标记
    // 检测叠加视频图像背景色的模式
    DETECT_TYPE      mDetectType;
    BOOL             mDetectOnceFinished;     // 指示检测一次完成
    BYTE             mBackColorR;             // 背景色的R分量
    BYTE             mBackColorG;             // 背景色的G分量
    BYTE             mBackColorB;             // 背景色的B分量
    BYTE             mBackDrift;              // 背景色的偏移值
    BOOL             mIsMMXSupported;         // 当前CPU是否支持MMX
public:
```

```
    CVideoMixController();
    ~CVideoMixController();
private:
    void ReleaseMixingBuffer(void);
    __inline BYTE MyAbs(__int16 inValue);
    // 各个抠像叠加算法实现（前文已经介绍）
    __inline void ActualMixing16Bits(BYTE * outPrimary, BYTE * inMixing);
    __inline void ActualMixing32Bits(BYTE * outPrimary, BYTE * inMixing);
    __inline void IdealMixing16Bits(BYTE * outPrimary, BYTE * inMixing);
    __inline void IdealMixing32Bits(BYTE * outPrimary, BYTE * inMixing);
    BOOL IsMMXSupported(void); // 检测当前CPU是否支持MMX
    void IdealMMXMixing16Bits(BYTE * outPrimary, BYTE * inMixing);
public:
    void Start(void);  // 开始视频叠加
    void Stop(void);   // 结束所有叠加
    // 设置输入图像的格式信息
    void SetInputVideoInfo(VIDEOINFOHEADER * inInfo);
    // IVideoKeyer接口委托过来的函数
    // 让外部设置/得到检测背景色的模式
    void SetDetectType(DETECT_TYPE inType);
    void GetDetectType(DETECT_TYPE * outType);
    // 外部设置/得到自定义的背景色
    void SetBackgroundColor(BYTE inRed, BYTE inGreen, BYTE inBlue);
    void GetBackgroundColor(BYTE * outRed, BYTE * outGreen, BYTE * outBlue);
    // Mixing In Pin“委托”过来的函数
    HRESULT ReceiveMixingVideo(IMediaSample * inMixingSample);
    HRESULT MixingEndOfStream(void);
    HRESULT MixingBeginFlush(void);
    HRESULT MixingEndFlush(void);
    // 外部调用该函数进行视频抠像叠加
    HRESULT DoMixing(IMediaSample * inMainSample);
    // 主视频流结束的通知
    HRESULT MainVideoEndOfStream(void);
};
#endif // __H_CVideoMixController__

//
// CVideoMixController.cpp
//

#include <streams.h>
#include <stdio.h>
#include "CVideoMixController.h"
//////////////////////////////////////////////////////////////////////////
CVideoMixController::CVideoMixController()
{
    // 参数初始化
    mMixingFrames        = NULL;
    mMixingFrameSize     = 0;
    mTotalPixels         = 0;
    mBitCount            = 0;
    mEOSMixing           = FALSE;
```

```
    mEOSMainVideo        = FALSE;
    mStopped             = FALSE;
    mMixingFlushing      = FALSE;
    mDetectType          = CUSTOMIZE;
    mDetectOnceFinished = FALSE;
    mBackColorR          = 0;
    mBackColorG          = 0;
    mBackColorB          = 0;
    mBackDrift           = DEFAULT_RGB565_DRIFT;
    // 检测当前CPU是否支持MMX指令集
    mIsMMXSupported = IsMMXSupported();
}
CVideoMixController::~CVideoMixController()
{
    ReleaseMixingBuffer();
}
// 销毁缓存对象
void CVideoMixController::ReleaseMixingBuffer(void)
{
    if (mMixingFrames)
    {
        delete mMixingFrames;
        mMixingFrames = NULL;
    }
}
// 开始视频叠加（由Filter开始运行时调用）
void CVideoMixController::Start(void)
{
    mEOSMixing           = FALSE;
    mEOSMainVideo        = FALSE;
    mStopped             = FALSE;
    mMixingFlushing      = FALSE;
    mMixingFrames->SetReachEnd(FALSE);
}
// 结束所有叠加（由Filter停止运行前调用）
void CVideoMixController::Stop(void)
{
    mStopped = TRUE;
}
// 设置输入图像的格式信息（由Filter上Input Pin完成连接时调用）
void CVideoMixController::SetInputVideoInfo(VIDEOINFOHEADER * inInfo)
{
    ASSERT(inInfo);
    mBitCount    = inInfo->bmiHeader.biBitCount;
    mTotalPixels = inInfo->bmiHeader.biWidth * abs(inInfo->bmiHeader.
        biHeight);
    // 图像宽度按4字节对齐
    long stride = (inInfo->bmiHeader.biWidth * (mBitCount / 8) + 3) & ~3;
    mMixingFrameSize = stride * abs(inInfo->bmiHeader.biHeight);
    mBackDrift  = (mBitCount == 16) ? DEFAULT_RGB565_DRIFT : DEFAULT_
        RGB32_DRIFT;
    // 创建缓存对象，用于缓存叠加视频图像帧
```

```
        ReleaseMixingBuffer();
        mMixingFrames = new CFrameList(mMixingFrameSize, BUFFERED_FRAME_
            COUNT);
    }
    // 外部设置一种检测背景色的模式
    void CVideoMixController::SetDetectType(DETECT_TYPE inType)
    {
        mDetectType = inType;
    }
    void CVideoMixController::GetDetectType(DETECT_TYPE * outType)
    {
        *outType = mDetectType;
    }
    // 外部设置自定义的背景色
    void CVideoMixController::SetBackgroundColor(BYTE inRed, BYTE inGreen, BYTE
        inBlue)
    {
        mBackColorR = inRed;
        mBackColorG = inGreen;
        mBackColorB = inBlue;
        // 同步改变背景色的检测模式
        mDetectType = CUSTOMIZE;
    }
    void CVideoMixController::GetBackgroundColor(BYTE * outRed, BYTE *
        outGreen, BYTE * outBlue)
    {
        *outRed     = mBackColorR;
        *outGreen   = mBackColorG;
        *outBlue    = mBackColorB;
    }
    // 接收到叠加视频图像数据后进行缓存
    HRESULT CVideoMixController::ReceiveMixingVideo(IMediaSample *
        inMixingSample)
    {
        if (mEOSMixing || mStopped || mMixingFlushing)
        {
            return NOERROR;
        }

        PBYTE  pData;
        inMixingSample->GetPointer(&pData);
        LONG length = inMixingSample->GetActualDataLength();
        if (mMixingFrames)
        {
            mMixingFrames->AddFrame(pData);
        }
        return NOERROR;
    }
    // 叠加视频流结束的通知
    HRESULT CVideoMixController::MixingEndOfStream(void)
    {
        mEOSMixing = TRUE;
```

```
    // 告诉缓存对象：不再有图像帧进入缓存
    mMixingFrames->SetReachEnd(TRUE);
    return  NOERROR;
}
// 叠加视频流开始“清场”
HRESULT CVideoMixController::MixingBeginFlush(void)
{
    mMixingFlushing = TRUE;
    if (mMixingFrames)
    {
        // 清空缓存对象中已经缓存的图像数据
        mMixingFrames->Flush();
    }
    return  NOERROR;
}
// 叠加视频流结束“清场”
HRESULT CVideoMixController::MixingEndFlush(void)
{
    mMixingFlushing = FALSE;
    return  NOERROR;
}
// Filter在接收到一个主视频图像帧后调用该函数进行一次抠像叠加
HRESULT CVideoMixController::DoMixing(IMediaSample * inMainSample)
{
    if (mEOSMainVideo || mStopped)
    {
        return NOERROR;
    }
    ASSERT(inMainSample);
    ASSERT(mMixingFrames);

    PBYTE  pSrc = NULL, pMixing = NULL;
    // 得到主视频图像数据
    inMainSample->GetPointer(&pSrc);
    // 从缓存对象中获取一帧叠加视频图像数据
    mMixingFrames->GetFrame(&pMixing);
    // 如果此时缓存中没有待叠加的图像，则跳过，不对主视频图像做任何处理
    if (pMixing != NULL)
    {
        // 进行视频抠像叠加
        if (mBitCount == 16)  // 16位RGB格式的叠加
        {
            // 优先使用MMX实现的算法
            if (mIsMMXSupported)
            {
                IdealMMXMixing16Bits(pSrc, pMixing);
            }
            else
            {
                IdealMixing16Bits(pSrc, pMixing);
            }
        }
```

```
        else  // 32位RGB格式的叠加
        {
            IdealMixing32Bits(pSrc, pMixing);
        }
        // 解锁缓存对象
        mMixingFrames->ReleaseFrame();
    }
    return NOERROR;
}
// 主视频流结束的通知
HRESULT CVideoMixController::MainVideoEndOfStream(void)
{
    mEOSMainVideo = TRUE;
    return NOERROR;
}
```

因为控制类CVideoMixController实现了几种叠加图像中的背景色检测模式供用户选择，因此可以在Filter上实现一个自定义接口。这个自定义接口声明如下：

```
//
// IVideoKeyer.h
//

#ifndef __H_IVideoKeyer__
#define __H_IVideoKeyer__

// 背景色检测模式定义
typedef enum
{
    DETECT_ONCE,     // 只在第一次叠加时检测一次
    DETECT_ALWAYS,   // 每次叠加都检测
    CUSTOMIZE        // 用户设定背景色
} DETECT_TYPE;

#ifdef __cplusplus
extern "C" {
#endif

//----------------------------------------------------------------------
// IVideoKeyer的各个接口函数定义
//----------------------------------------------------------------------
DECLARE_INTERFACE_(IVideoKeyer, IUnknown)
{
    // 得到输入Pin连接用的媒体类型
    STDMETHOD(get_ConnectedMediaType) (THIS_
        CMediaType * outMainType, CMediaType * outMixingType
    ) PURE;
    // 设置背景色检测模式
    STDMETHOD(put_DetectType) (THIS_
        DETECT_TYPE inType
    ) PURE;
    // 得到当前使用的背景色检测模式
```

```
    STDMETHOD(get_DetectType) (THIS_
        DETECT_TYPE * outType
    ) PURE;
    // 设置自定义的背景色
    STDMETHOD(put_BackgroundColor) (THIS_
        BYTE inRed, BYTE inGreen, BYTE inBlue
    ) PURE;
    // 得到当前使用的自定义背景色
    STDMETHOD(get_BackgroundColor) (THIS_
        BYTE * outRed, BYTE * outGreen, BYTE * outBlue
    ) PURE;
};
//-----------------------------------------------------------------------

#ifdef __cplusplus
}
#endif

#endif // __H_IVideoKeyer__
```

IVideoKeyer接口在Filter上的实现如下：

```
//
// CFilterVideoKeyer.h
//

#ifndef __H_CFilterVideoKeyer__
#define __H_CFilterVideoKeyer__

#include "IVideoKeyer.h"

class CFilterVideoKeyer : public CTransInPlaceFilter, public IVideoKeyer
{
public:
    DECLARE_IUNKNOWN;
    // 用于在Filter上暴露IVideoKeyer接口
    STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void ** ppv);
    // 其他成员（省略）
    // ……

    // --- IVideoKeyer接口函数声明 ---
    STDMETHODIMP get_ConnectedMediaType(CMediaType * outMainType,
        CMediaType * outMixingType);
    STDMETHODIMP put_DetectType(DETECT_TYPE inType);
    STDMETHODIMP get_DetectType(DETECT_TYPE * outType);
    STDMETHODIMP put_BackgroundColor(BYTE inRed, BYTE inGreen, BYTE
        inBlue);
    STDMETHODIMP get_BackgroundColor(BYTE * outRed, BYTE * outGreen, BYTE
        * outBlue);
};

#endif // __H_CFilterVideoKeyer__
```

```
//
// CFilterVideoKeyer.cpp
//
// 在Filter上暴露IVideoKeyer接口
STDMETHODIMP CFilterVideoKeyer::NonDelegatingQueryInterface(REFIID riid,
    void ** ppv)
{
    CheckPointer(ppv, E_POINTER);
    if (riid == IID_IVideoKeyer)
    {
        return GetInterface((IVideoKeyer *) this, ppv);
    }
    else
    {
        return CTransInPlaceFilter::NonDelegatingQueryInterface(riid,
            ppv);
    }
}
// --- IVideoKeyer的各接口函数实现 ---
// 得到输入Pin连接用的媒体类型
STDMETHODIMP CFilterVideoKeyer::get_ConnectedMediaType(CMediaType *
    outMainType, CMediaType * outMixingType)
{
    HRESULT hr = S_OK;
    if (outMainType)
    {
        // 检查Input Pin是否已经连接
        if (m_pInput && m_pInput->IsConnected())
        {
            *outMainType = m_pInput->CurrentMediaType();
        }
        else
        {
            hr = E_FAIL;
        }
    }
    if (SUCCEEDED(hr) && outMixingType)
    {
        // 检查Mixing In Pin是否已经连接
        if (mMixingInput && mMixingInput->IsConnected())
        {
            *outMixingType = mMixingInput->CurrentMediaType();
        }
        else
        {
            hr = E_FAIL;
        }
    }
    return hr;
}
// 设置叠加图像中背景色的检测模式
STDMETHODIMP CFilterVideoKeyer::put_DetectType(DETECT_TYPE inType)
```

```
{
    // 仅在Filter处于停止状态时支持检测模式的改变
    if (m_State == State_Stopped)
    {
        mController.SetDetectType(inType);
        return NOERROR;
    }
    return E_FAIL;
}
// 得到叠加图像中背景色的检测模式
STDMETHODIMP CFilterVideoKeyer::get_DetectType(DETECT_TYPE * outType)
{
    mController.GetDetectType(outType);
    return NOERROR;
}
// 设置自定义的背景色
STDMETHODIMP CFilterVideoKeyer::put_BackgroundColor(BYTE inRed, BYTE
    inGreen, BYTE inBlue)
{
    // 仅在Filter处于停止状态时支持自定义背景色的改变
    if (m_State == State_Stopped)
    {
        mController.SetBackgroundColor(inRed, inGreen, inBlue);
        return NOERROR;
    }
    return E_FAIL;
}
// 得到当前使用的自定义背景色
STDMETHODIMP CFilterVideoKeyer::get_BackgroundColor(BYTE * outRed, BYTE *
    outGreen, BYTE * outBlue)
{
    mController.GetBackgroundColor(outRed, outGreen, outBlue);
    return NOERROR;
}
```

提示：本书配套光盘的SourceCodes\Chapter04\VideoKey\FilterVideoKeyer目录下提供了视频抠像Filter的完整实现。

4.1.5 模拟实时源

HQ Video Keyer Filter开发完成之后，就可以进行视频抠像叠加的演示了。不过，需要一个能够产生单一背景的视频流的源！或许你手头并没有视频采集设备，没关系，使用SDK自带的一个Filter例子Ball（位于SDK目录的Samples\C++\DirectShow\Filters\Ball下）可以模拟实时源——这个Filter可以产生单一背景下弹球运动轨迹的视频流。

因为HQ Video Keyer要求输入的两路图像尺寸相同，因此需要给Ball Filter增加一个属性页，通过属性页我们可以定制Ball Filter输出图像的尺寸。（注意：Ball Filter的注册名字同时也改成了Bouncing Ball plus。）首先要定义一个如下的接口：

```
//
```

```
// IBall.h
//

#ifndef __H_IBall__
#define __H_IBall__

// 标识IBall接口的GUID
// {A1BBB8F4-7DDF-45be-8386-C492F167BC81}
DEFINE_GUID(IID_IBall, 0xa1bbb8f4,
    0x7ddf, 0x45be, 0x83, 0x86, 0xc4, 0x92, 0xf1, 0x67, 0xbc, 0x81);

#ifdef __cplusplus
extern "C" {
#endif

DECLARE_INTERFACE_(IBall, IUnknown)
{
    // 设置输出图像的尺寸
    STDMETHOD(SetImageSize) (THIS_
        int inWidth, int inHeight
    ) PURE;
    // 得到输出图像的尺寸
    STDMETHOD(GetImageSize) (THIS_
        int * outWidth, int * outHeight
    ) PURE;
};

#ifdef __cplusplus
}
#endif

#endif // __H_IBall__
```

然后在Filter上实现IBall接口（为了让定制的图像尺寸同时保存到GRF文件中去，Ball Filter同时实现一个IPersistStream接口）：

```
//
// fball.h
//

#include "IBall.h"

class CBouncingBall : public CSource, public IBall, public CPersistStream
{
public:
    // 其他成员声明（省略）
    // ……
    STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void ** ppv);
    // --- IBall接口方法声明---
    STDMETHODIMP SetImageSize(int inWidth, int inHeight);
    STDMETHODIMP GetImageSize(int * outWidth, int * outHeight);
    // CPersistStream类的纯虚函数
```

```
    HRESULT WriteToStream(IStream *pStream);
    HRESULT ReadFromStream(IStream *pStream);
    int SizeMax();
    STDMETHODIMP GetClassID(CLSID *pClsid);
};

//
// fball.cpp
//
// Filter类的构造函数
CBouncingBall::CBouncingBall(LPUNKNOWN lpunk, HRESULT *phr) :
CSource(NAME("Bouncing ball"), lpunk, CLSID_BouncingBall),
CPersistStream(lpunk, phr) // 对父类CPersistStream进行初始化
{
    // 代码省略
    // ……
}
// 在Filter上暴露IBall和IPersistStream两个接口
STDMETHODIMP CBouncingBall::NonDelegatingQueryInterface(REFIID riid, void
    ** ppv)
{
    CheckPointer(ppv, E_POINTER);
    if (riid == IID_IBall)
    {
        return GetInterface((IBall *) this, ppv);
    }
    else if (riid == IID_IPersistStream)
    {
        return GetInterface((IPersistStream *) this, ppv);
    }
    else
    {
        return CSource::NonDelegatingQueryInterface(riid, ppv);
    }
}
// --- IBall接口方法实现---
// 设置输出图像的尺寸
STDMETHODIMP CBouncingBall::SetImageSize(int inWidth, int inHeight)
{
    // 仅当Filter处在停止状态、并且输出Pin断开的情况下支持输出图像尺寸的改变
    if (IsStopped() && m_paStreams[0]->IsConnected() == FALSE)
    {
        ((CBallStream*)m_paStreams[0])->SetImageSize(inWidth, inHeight);
        return S_OK;
    }
    return E_FAIL;
}
// 得到输出图像的尺寸
STDMETHODIMP CBouncingBall::GetImageSize(int * outWidth, int * outHeight)
{
    ((CBallStream*)m_paStreams[0])->GetImageSize(outWidth, outHeight);
    return NOERROR;
```

```
}
// 实现CPersistStream类的4个纯虚函数
// 将图像的尺寸保存
HRESULT CBouncingBall::WriteToStream(IStream *pStream)
{
    int width = 0, height = 0;
    GetImageSize(&width, &height);
    HRESULT hr = WriteInt(pStream, width);
    hr = WriteInt(pStream, height);
    return hr;
}
// 读出图像的尺寸
HRESULT CBouncingBall::ReadFromStream(IStream *pStream)
{
    HRESULT hr  = NOERROR;
    int  width  = ReadInt(pStream, hr);
    int  height = ReadInt(pStream, hr);
    SetImageSize(width, height);
    return hr;
}
int CBouncingBall::SizeMax()
{
    return (24 * 2);
}
STDMETHODIMP CBouncingBall::GetClassID(CLSID *pClsid)
{
    return CBaseFilter::GetClassID(pClsid);
}

//
// fball.h
//
class CBallStream : public CSourceStream
{
public:
    // 其他成员声明（省略）
    // ……
    // 在Pin类上增加两个公有接口用于调整输出图像的尺寸
    void SetImageSize(int inWidth, int inHeight);
    void GetImageSize(int * outWidth, int * outHeight);
};

//
// fball.cpp
//
void CBallStream::SetImageSize(int inWidth, int inHeight)
{
m_iImageWidth  = inWidth;
m_iImageHeight = inHeight;
// 对负责绘制弹球的控制对象进行同步更新
    if (m_Ball)
    {
```

```
            m_Ball->SetImageSize(m_iImageWidth, m_iImageHeight);
        }
    }
    void CBallStream::GetImageSize(int * outWidth, int * outHeight)
    {
        if (outWidth && outHeight)
        {
            *outWidth  = m_iImageWidth;
            *outHeight = m_iImageHeight;
        }
    }

    //
    // ball.h
    //
    class CBall  // 这个类负责弹球的绘制
    {
    public:
        // 其他成员声明（省略）
        // ……
        // 增加用于调整输出图像的尺寸的公有函数
        void SetImageSize(int inWidth, int inHeight);
    };

    //
    // ball.cpp
    //
    void CBall::SetImageSize(int inWidth, int inHeight)
    {
        m_iImageWidth  = inWidth;
        m_iImageHeight = inHeight;
    }
```

最后给Ball Filter增加一个属性页。首先在VC的资源编辑器中添加一个对话框，注意将该对话框的属性Style设成Child，Border设成None。然后添加必要的控件，编辑如图4.3所示。

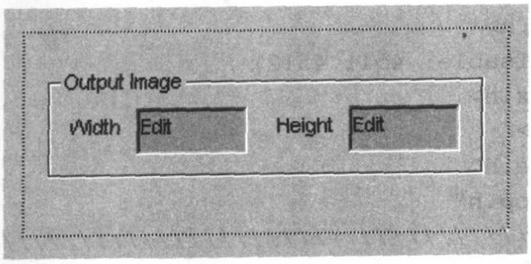


图4.3 Ball Filter的属性页

定义一个属性页类CBallProp从CBasePropertyPage类派生，并且实现一些必要的函数：

```
    //
    // CBallProp.h
    //
```

```
#ifndef __H_CBallProp__
#define __H_CBallProp__

#include "IBall.h"

class CBallProp : public CBasePropertyPage
{
private:
    IBall *      mIBall;      // 从Filter对象上获得的IBall接口
    // 属性页上的控件窗口句柄
    HWND         mEditWidth;
    HWND         mEditHeight;
public:
    static CUnknown * WINAPI CreateInstance(LPUNKNOWN lpunk, HRESULT *phr);
    DECLARE_IUNKNOWN;
private:
    CBallProp(LPUNKNOWN lpunk, HRESULT *phr);
private:
    BOOL OnReceiveMessage(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam);
    HRESULT OnConnect(IUnknown *pUnknown);
    HRESULT OnDisconnect();
    HRESULT OnActivate();
    HRESULT OnApplyChanges();
    void SetDirty();
private:
    void ReflectImageSize(void);
    void EnterImageSize(void);
};

#endif // __H_CBallProp__

//
// CBallProp.cpp
//

#include <streams.h> .
#pragma warning(disable: 4511 4512)
#include <windowsx.h>
#include <stdio.h>
#include "Resource.h"
#include "CBallProp.h"
//////////////////////////////////////////////////////////////////////////
// 创建属性页组件对象
CUnknown * WINAPI CBallProp::CreateInstance(LPUNKNOWN lpunk, HRESULT *phr)
{
    CUnknown *punk = new CBallProp(lpunk, phr);
    if (punk == NULL)
    {
        *phr = E_OUTOFMEMORY;
    }
```

```
    return punk;
}
// 属性页类的构造函数
CBallProp::CBallProp(LPUNKNOWN pUnk, HRESULT *phr) :
CBasePropertyPage(NAME("Prop Page"),pUnk,IDD_DLG_PROP,IDS_TITLE)
{
    ASSERT(phr);
    mIBall = NULL;
}
// 属性页窗口的消息处理函数
BOOL CBallProp::OnReceiveMessage(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM
    lParam)
{
    switch (uMsg)
    {
    case WM_INITDIALOG: // 对话框初始化消息
    {
        // 获得各控件的窗口句柄
        mEditWidth  = GetDlgItem(hwnd, IDC_EDIT_IMAGE_WIDTH);
        mEditHeight  = GetDlgItem(hwnd, IDC_EDIT_IMAGE_HEIGHT);
        break;
    }
    case WM_COMMAND:
    {
        SetDirty();
        break;
    }

    }
    return CBasePropertyPage::OnReceiveMessage(hwnd,uMsg,wParam,lParam);
}
// 与Filter建立连接
HRESULT CBallProp::OnConnect(IUnknown *pUnknown)
{
    // 从Filter对象上获得IBall接口
    HRESULT hr = pUnknown->QueryInterface(IID_IBall, (void **) &mIBall);
    if (FAILED(hr))
    {
        return E_NOINTERFACE;
    }
    ASSERT(mIBall);
    return NOERROR;
}
// 与Filter断开连接
HRESULT CBallProp::OnDisconnect()
{
    // 释放IBall接口
    if (mIBall == NULL)
        return E_UNEXPECTED;
    mIBall->Release();
```

```
        mIBall = NULL;
        return NOERROR;
    }
    // 属性页被激活
    HRESULT CBallProp::OnActivate()
    {
        // 显示当前Filter上的输出图像尺寸
        ReflectImageSize();
        return NOERROR;
    }
    // 属性页上的参数已改变（使能Apply按钮）
    void CBallProp::SetDirty()
    {
        m_bDirty = TRUE;
        if (m_pPageSite)
        {
            m_pPageSite->OnStatusChange(PROPPAGESTATUS_DIRTY);
        }
    }
    // Apply按钮的响应函数——使参数生效
    HRESULT CBallProp::OnApplyChanges()
    {
        // 将新的图像尺寸设置给Filter
        EnterImageSize();
        return NOERROR;
    }
    // 获取当前Filter上的输出图像尺寸，并在属性页上显示
    void CBallProp::ReflectImageSize(void)
    {
        if (mIBall)
        {
            int width = 0, height = 0;
            mIBall->GetImageSize(&width, &height);
            char value[100];
            sprintf(value, "%d", width);
            SetWindowText(mEditWidth, value);
            sprintf(value, "%d", height);
            SetWindowText(mEditHeight, value);
        }
    }
    // 将新的图像尺寸设置给Filter
    void CBallProp::EnterImageSize(void)
    {
        if (mIBall)
        {
            char strWidth[10], strHeight[10];
            int ret1 = GetWindowText(mEditWidth, strWidth, 10);
            int ret2 = GetWindowText(mEditHeight, strHeight, 10);
            if (ret1 && ret2)
            {
```

```
            mIBall->SetImageSize(atoi(strWidth), atoi(strHeight));
        }
    }
}
```

另外，属性页也是一种COM组件，需要进行注册。因此，必须给它填写注册信息，并且在Filter上实现一个ISpecifyPropertyPages接口。

```
// Ball Filter属性页的CLSID定义（使用系统的GUIDGEN.EXE生成）
// {AC246CF2-EBCB-4f94-BD22-BE4DD207A64A}
DEFINE_GUID(CLSID_BouncingBallProp, 0xac246cf2, 0xebcb,
    0x4f94, 0xbd, 0x22, 0xbe, 0x4d, 0xd2, 0x7, 0xa6, 0x4a);

//
// fball.h
//
class CBouncingBall : public CSource, public ISpecifyPropertyPages
{
public:
    // 其他成员声明（省略）
    // ……
    // --- ISpecifyPropertyPages接口方法声明 ---
    STDMETHODIMP GetPages(CAUUID *pPages);
};

//
// fball.cpp
//
CFactoryTemplate g_Templates[] =
{
    {
        L"Bouncing Ball plus",    // Ball Filter新的注册名字
        &CLSID_BouncingBall,
        CBouncingBall::CreateInstance,
        NULL,
        &sudBallax
    },
    {
        L"Ball Property Page",
        &CLSID_BouncingBallProp,
        CBallProp::CreateInstance
    }
};
int g_cTemplates = sizeof(g_Templates) / sizeof(g_Templates[0]);

// 暴露ISpecifyPropertyPages接口
STDMETHODIMP CBouncingBall::NonDelegatingQueryInterface(REFIID riid, void
    ** ppv)
{
    CheckPointer(ppv, E_POINTER);
    if (riid == IID_IBall)
```

```
        {
            return GetInterface((IBall *) this, ppv);
        }
        else if (riid == IID_ISpecifyPropertyPages)
        {
            return GetInterface((ISpecifyPropertyPages *) this, ppv);
        }
        else if (riid == IID_IPersistStream)
        {
            return GetInterface((IPersistStream *) this, ppv);
        }
        else
        {
            return CSource::NonDelegatingQueryInterface(riid, ppv);
        }
}
// --- ISpecifyPropertyPages接口方法实现 ---
STDMETHODIMP CBouncingBall::GetPages(CAUUID *pPages)
{
    pPages->cElems = 1;
    pPages->pElems = (GUID *) CoTaskMemAlloc(sizeof(GUID));
    if (pPages->pElems == NULL)
    {
        return E_OUTOFMEMORY;
    }
    *(pPages->pElems) = CLSID_BouncingBallProp;
    return NOERROR;
}
```

提示：本书配套光盘的SourceCodes\Chapter04\VideoKey\Ball目录下提供了扩展版本的Ball Filter的完整实现。

4.1.6 应用演示

终于可以开始演示了！运行GraphEdit（即SDK目录下的\Bin\DXUtils\graphedt.exe），首先选择一个媒体文件构建如图4.4所示的Filter Graph。

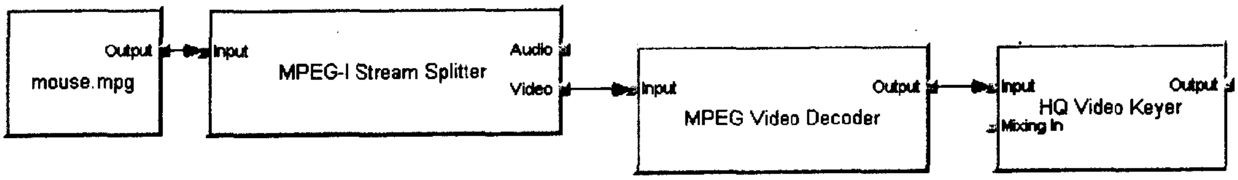


图 4.4 抠像叠加 Filter Graph 之步骤 1

然后打开HQ Video Keyer Filter的属性页（右击该Filter，在弹出菜单中选择Filter Properties命令即可），查看HQ Video Keyer的Input Pin上输入图像的尺寸（从图4.5中可以看到图像的尺寸为320 x 240）。

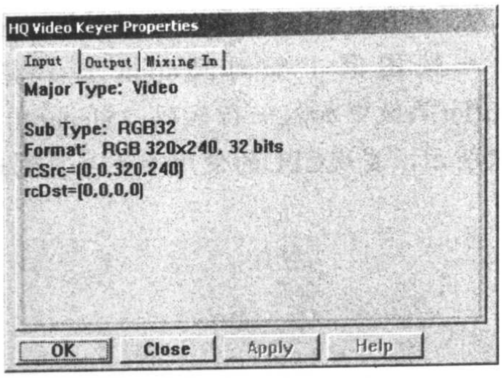


图 4.5　HQ Video Keyer Filter 的属性页

然后在GraphEdit中插入一个Bouncing Ball plus Filter，随后打开它的属性页，设置输出图像尺寸与HQ Video Keyer的Input Pin保持一致（也设成320 x 240）。接着将Bouncing Ball plus与HQ Video Keyer的Mixing In Pin相连，再将HQ Video Keyer的输出Pin连接到一个Video Renderer。最终构建的Filter Graph如图4.6所示。

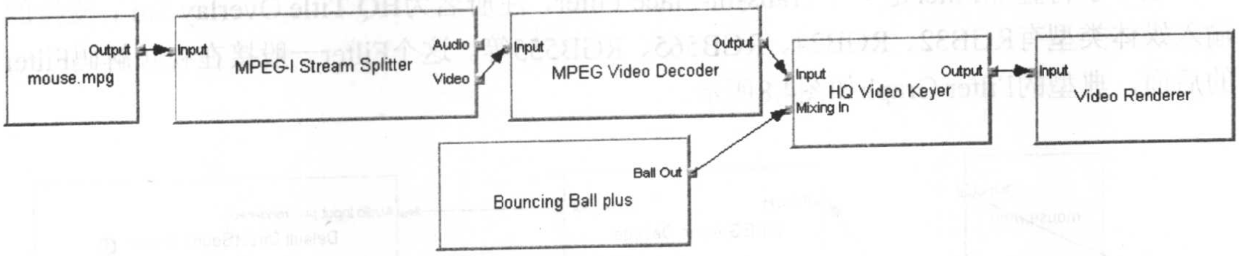


图 4.6　抠像叠加 Filter Graph 之步骤 2

最后运行这个Filter Graph，就可以看到如图4.7所示的视频效果了（主视频上叠加了一个到处乱撞的弹球）。

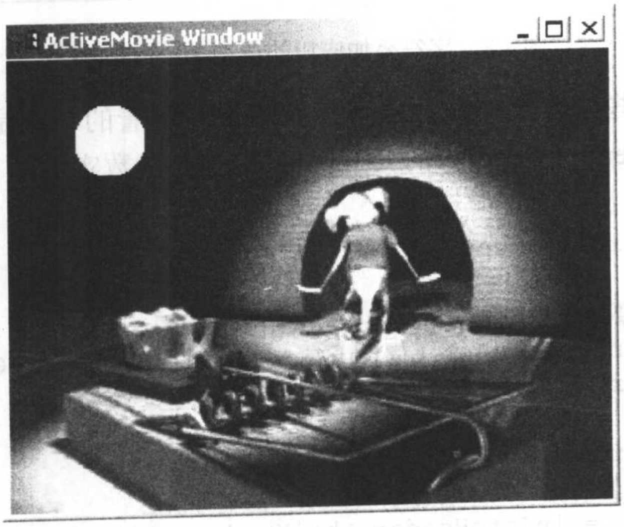


图 4.7　视频抠像叠加的预览效果

提示：演示中，你可能会发现CPU的使用率居然持续高达100%。为什么呢？经过仔细追踪后发现，Ball Filter的推数据线程一直处于全速工作状态，抢占

了很多CPU资源。为了让Ball Filter更像一个实时源，应该让Ball Filter每隔一定的时间才输出一帧图像（如间隔40ms）。实现方法很简单，在CBallStream::FillBuffer函数中加入一行代码：Sleep(40);，然后重新编译并注册Ball Filter。再次演示，发现CPU的使用率立竿见影地降到了40%以下（这是一个正常水平）。

4.2 字符叠加

4.2.1 开发字符叠加Filter

《DirectShow开发指南》一书对字符叠加Filter的开发作了比较详细的介绍。字符叠加的实现原理这里就不再重复介绍了。只是在这个Filter的使用和扩展方面，还有必要进行一些提示（很多读者都在这方面遇到了麻烦）。

这个字符叠加Filter是一个Trans-In-Place Filter，注册名为HQ Title Overlay Std.，支持的输入媒体类型有RGB32、RGB24、RGB565、RGB555等。这个Filter一般接在视频解码Filter的后面，典型的Filter Graph如图4.8所示。

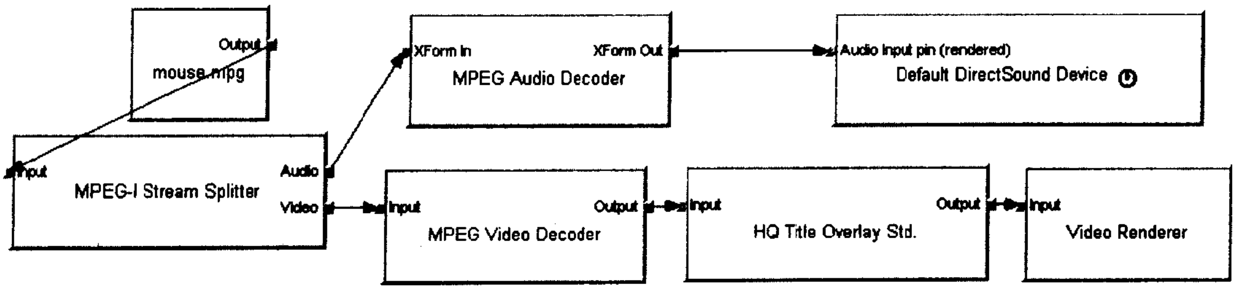


图 4.8　字符叠加应用的典型 Filter Graph

实际应用中，有些读者反映叠加的字符有时候是倒着的！这是源程序中存在的一个Bug。参见控制类COverlayController的SetInputVideoInfo函数实现：

```
//
// COverlayController.cpp
//
// 将输入Pin上将要接收的图像的格式信息设置给控制对象
void COverlayController::SetInputVideoInfo(const VIDEOINFOHEADER *
    inInfoHeader)
{
    ASSERT(inInfoHeader);
    // 获取输入图像的宽、高、每像素的位数、帧率等信息
    mImageWidth  = inInfoHeader->bmiHeader.biWidth;
    mImageHeight = (DWORD)(abs(inInfoHeader->bmiHeader.biHeight)); // ?
    mImageBitCount = inInfoHeader->bmiHeader.biBitCount;
    if (inInfoHeader->AvgTimePerFrame > 0)
    {
```

```
        mInputFrameRate = 1.0 * UNITS / inInfoHeader->AvgTimePerFrame;
        SideEffectProgressChanged();
    }
    // 计算图像的宽度（以字节为单位，并且按照4字节对齐）
    if (mImageBitCount != 0 && 0 == (7 & mImageBitCount))
    {
        mImageWidthInBytes = (mImageWidth * (mImageBitCount / 8) + 3) & ~3;
    }
    else
    {
        mImageWidthInBytes = mImageWidth;
    }
    mIsBottomUpImage = mImageHeight > 0 ? TRUE : FALSE; // ?
}
```

看出问题了吗？inInfoHeader->bmiHeader.biHeight取绝对值之后赋给变量mImageHeight，那么mImageHeight的值永远是正的，于是最后一行代码根据mImageHeight的正负决定输入图像的扫描方式就变得没有意义了。大家知道，图像有两种扫描方式：从下往上和从上往下。前者多用于GDI中，后者多用于DirectDraw中。图像的扫描方式通过BITMAPINFOHEADER的成员biHeight的正负来区别：正数表示从下往上，负数表示从上往下。因为字符位图是通过GDI函数创建的，它的数据肯定是从下往上扫描存储的；而如果输入Pin上接收的图像是从上往下扫描的，上述SetInputVideoInfo函数的实现会误认为图像是从下往上扫描的，于是在字符的实际叠加过程中就会将字符内容叠加成倒的（参见COverlayController::ActualOverlay函数）。

提示：更多关于图像扫描方式的介绍，请读者参考2.2.9节。

因此，COverlayController::SetInputVideoInfo函数必须作出如下修正：

```
void COverlayController::SetInputVideoInfo(const VIDEOINFOHEADER *
    inInfoHeader)
{
    ASSERT(inInfoHeader);
    mImageWidth     = inInfoHeader->bmiHeader.biWidth;
    mImageHeight    = (DWORD)(abs(inInfoHeader->bmiHeader.biHeight));
    mImageBitCount  = inInfoHeader->bmiHeader.biBitCount;
    if (inInfoHeader->AvgTimePerFrame > 0)
    {
        mInputFrameRate = 1.0 * UNITS / inInfoHeader->AvgTimePerFrame;
        SideEffectProgressChanged();
    }
    if (mImageBitCount != 0 && 0 == (7 & mImageBitCount))
    {
        mImageWidthInBytes = (mImageWidth * (mImageBitCount / 8) + 3) & ~3;
    }
    else
    {
        mImageWidthInBytes = mImageWidth;
    }
```

```
    ///////////////////////////// 修正 ////////////////////////////////////
    mIsBottomUpImage = inInfoHeader->bmiHeader.biHeight > 0 ? TRUE : FALSE;
}
```

另外，有些读者想要对这个字符叠加Filter进行扩展，如支持叠加的字符内容动态改变。虽然ITitleOverlay接口的put_Title函数能够改变字符的内容，但是在Filter Graph运行的时候调用这个接口方法将会返回一个错误码，叠加的字符内容也不会改变。为什么呢？我们来看put_Title函数的实现：

```
//
// CFilterTitleOverlay.cpp
//
STDMETHODIMP CFilterTitleOverlay::put_Title(const char * inTitle, int
    inLength)
{
    CAutoLock   lockit(&mITitleOverlaySync);
    // 仅当Filter处于停止状态时支持字符内容的改变
    if (m_State != State_Stopped)
    {
        return E_UNEXPECTED;
    }
    mOverlayController->SetTitle(inTitle, inLength);
    return NOERROR;
}
```

这个字符叠加Filter仅在Filter处于停止状态时才支持字符内容的改变。为什么要加这个限制呢？简单地将上述限制条件注释掉，然后再试，结果看到叠加的字符内容在Filter Graph运行时还是不能动态改变。怎么回事呢？这就要到控制类COverlayController中去找原因了。原来，字符位图数据仅在Filter Graph启动的时候生成一次（参见COverlayController::StartTitleOverlay函数），而在Filter Graph运行过程中更改字符内容后，Filter内部的字符位图数据并不会自动进行更新！所以，我们看到的视频图像上叠加的字符内容自然也不会改变。其实，要支持字符内容的动态改变也很简单，在字符内容修改完之后同步刷新控制对象内的字符位图数据就可以了。有兴趣的读者可以把这个功能的实现当作练习来完成。

还有些读者在VC中编译Filter项目的时候碰到如下问题（有时在开发需要连接DirectShow基类库的应用程序时也会碰到这样的问题）：

```
strmbasd.lib(wxutil.obj) : error LNK2005: "public: __thiscall
    CCritSec::CCritSec(void)" (??0CCritSec@@QAE@XZ) already defined in
    CFilterTitleOverlay.obj
strmbasd.lib(wxutil.obj) : error LNK2005: "public: __thiscall
    CCritSec::~CCritSec(void)" (??1CCritSec@@QAE@XZ) already defined in
    CFilterTitleOverlay.obj
strmbasd.lib(wxutil.obj) : error LNK2005: "public: void __thiscall
    CCritSec::Lock(void)" (?Lock@CCritSec@@QAEXXZ) already defined in
    CFilterTitleOverlay.obj
strmbasd.lib(wxutil.obj) : error LNK2005: "public: void __thiscall
    CCritSec::Unlock(void)" (?Unlock@CCritSec@@QAEXXZ) already defined in
    CFilterTitleOverlay.obj
```

```
strmbasd.lib(wxutil.obj) : warning LNK4006: "public: __thiscall
    CCritSec::CCritSec(void)" (??0CCritSec@@QAE@XZ) already defined in
    CFilterTitleOverlay.obj; second definition ignored
strmbasd.lib(wxutil.obj) : warning LNK4006: "public: __thiscall
    CCritSec::~CCritSec(void)" (??1CCritSec@@QAE@XZ) already defined in
    CFilterTitleOverlay.obj; second definition ignored
strmbasd.lib(wxutil.obj) : warning LNK4006: "public: void __thiscall
    CCritSec::Lock(void)" (?Lock@CCritSec@@QAEXXZ) already defined in
    CFilterTitleOverlay.obj; second definition ignored
strmbasd.lib(wxutil.obj) : warning LNK4006: "public: void __thiscall
    CCritSec::Unlock(void)" (?Unlock@CCritSec@@QAEXXZ) already defined in
    CFilterTitleOverlay.obj; second definition ignored
```

这主要是由VC定义的Debug环境与SDK定义的Debug环境不兼容引起的。前者通过定义一个_DEBUG宏来标识，而后者通过定义另外一个DEBUG宏来标识。如果使用9.0版本以前的SDK，编译Debug版本的程序时需要在VC的项目设置中确认DEBUG宏的定义，方法是，执行VC的菜单命令Project | Settings，然后在弹出的对话框中进入C/C++一页，Category一项选择“Preprocessor”，Preprocessor definitions下面确认定义了“DEBUG”（如果没有就追加一个）。而如果使用的是9.0版本的SDK，则不用这么麻烦了，SDK基类库在streams.h文件中对这个问题进行了修正，实现如下：

```
// File: Streams.h
//
// 如果检测到_DEBUG已经被定义，说明这是一个Debug环境，进一步检测DEBUG是否已经被定义，
// 如果没有则在这里定义一下
#ifdef _DEBUG
#ifndef DEBUG
#define DEBUG
#endif
#endif
```

提示：本书配套光盘的SourceCodes\Chapter04\TitleOverlay\FilterTitleOverlay目录下提供了更新版的字符叠加Filter的完整实现。

4.2.2 与媒体播放集成

《DirectShow开发指南》一书的第5章介绍了一个简单的媒体文件播放器，有些读者反映，不知如何将这个播放器与字符叠加Filter相集成。下面，我们就来解决这个问题。

提示：关于这个播放器的开发过程参见《DirectShow开发指南》一书的5.9节。

要在播放器中使用字符叠加Filter，首先要知道该Filter的CLSID（定义在FltTitleGuids.h文件中）。然后在应用程序中通过包含initguid.h头文件来将这些CLSID定义转换成GUID常量形式，实现如下（注意initguid.h必须放在FltTitleGuids.h的前面）：

```
// SimplePlayerDlg.cpp
//
```

```
#include "stdafx.h"
#include "SimplePlayer.h"
#include "SimplePlayerDlg.h"

#include <initguid.h>
#include "FltTitleGuids.h"
#include "ITitleOverlay.h"
#include "OverlayDefs.h"
```

提示：更多关于GUID的定义和使用方法，参见附录A的第31个问答“DEFINE_GUID这个宏是怎么工作的？”。

使用字符叠加Filter的Filter Graph应该构建成如图4.8一样。如何来构建呢？如果播放媒体文件的Filter链路是通过调用IFilterGraph::ConnectDirect一步一步构建的，那么在Filter链路中插入字符叠加Filter应该很容易。但如果使用的是智能连接方法，插入我们的字符叠加Filter就有些麻烦了——首先要找到Video Renderer，甚至是Overlay Mixer（在Overlay Mixer与Video Renderer配对使用的情况下，Overlay Mixer进行实际的图像显示，Video Renderer只是一个视频窗口管理器，因此应把字符叠加Filter插入到Overlay Mixer的前面），将原来的Pin连接断开，插入我们的字符叠加Filter，然后重新连接（将字符叠加Filter嵌在中间）。不幸的是，我们的简单媒体文件播放器使用的就是智能连接方法。因此，这个播放器程序需要修改如下：

```
// SimplePlayerDlg.cpp
//
// 构建Filter Graph
void CSimplePlayerDlg::CreateGraph(void)
{
    DestroyGraph();
    mFilterGraph = new CDXGraph();
    if (mFilterGraph->Create())
    {
        // 使用智能连接构建Filter Graph
        mFilterGraph->RenderFile(mSourceFile);
        /////////// 插入字符叠加Filter /////////////
        AddTitleOverlayFilter();
        ///////////////////////////////////////////////
        // 设置视频显示窗口和消息通知窗口
        mFilterGraph->SetDisplayWindow(mVideoWindow.GetSafeHwnd());
        mFilterGraph->SetNotifyWindow(this->GetSafeHwnd());
        // 显示视频的第一帧图像
        mFilterGraph->Pause();
    }
}
// 向已经构建完成的Filter Graph中插入一个字符叠加Filter
void CSimplePlayerDlg::AddTitleOverlayFilter(void)
{
    // 创建字符叠加Filter
    IBaseFilter * pFilter = NULL;
```

```
        HRESULT hr = CoCreateInstance(CLSID_HQTitleOverlay, NULL,
            CLSCTX_INPROC_SERVER,IID_IBaseFilter, (void **)&pFilter);
        if (SUCCEEDED(hr))
        {
            // 获得字符叠加Filter上的ITitleOverlay接口，然后调用ITitleOverlay的接口
            // 方法进行一系列的设置
            ITitleOverlay * pOverlayer = NULL;
            pFilter->QueryInterface(IID_ITitleOverlay, (void**)&pOverlayer);
            if (pOverlayer)
            {
                pOverlayer->put_TitleOverlayType(OT_SCROLL_BOTTOM);
                UpdateData();
                pOverlayer->put_Title(mOverlayTitle,
                    mOverlayTitle.GetLength());
                pOverlayer->put_TitleColor(0xff, 0, 0);
                pOverlayer->put_TitleDuration(0, 20);
                pOverlayer->Release();
            }
            // 将字符叠加Filter加入到Filter Graph中
            mFilterGraph->GetGraph()->AddFilter(pFilter, L"Title Overlay");
            // 字符叠加Filter加入到Filter Graph后，Filter Graph Manager会对它增加
            // 引用计数，因此应用程序此时将接口释放是安全的
            pFilter->Release();
            // 在Filter Graph中寻找连接到Video Renderer上的输出Pin
            IPin * pOut = FindPin_ConnectedToVideoRenderer();
            // 得到字符叠加Filter的输入Pin
            IPin * pIn  = GetPin(pFilter, TRUE);
            // 将字符叠加Filter接入
            BOOL pass = mFilterGraph->ConnectFilters(pOut, pIn);
            if (pass)
            {
                // 得到字符叠加Filter的输出Pin
                pOut = GetPin(pFilter, FALSE);
                // 使用智能连接方法连接字符叠加Filter后面的Filter链路
                mFilterGraph->GetGraph()->Render(pOut);
            }
            else
            {
                // 如果字符叠加Filter接入失败，则将最初构建的Filter Graph恢复
                mFilterGraph->GetGraph()->Render(pOut);
                AfxMessageBox("Title overlay filter connection failed!");
            }
        }
    }
    // 在Filter Graph中寻找连接到Video Renderer上的输出Pin
    IPin * CSimplePlayerDlg::FindPin_ConnectedToVideoRenderer()
    {
        IPin * foundPin = NULL;
        // 因为Video Renderer上实现了IVideoWindow接口，可以查看Filter上是否支持
        // IVideoWindow接口来定位Video Renderer
```

```
    IBaseFilter* pVR = NULL;
    HRESULT hr = FindFilterByInterface(IID_IVideoWindow, &pVR);
    if (SUCCEEDED(hr))
    {
        IBaseFilter * pOverlayMixer = NULL;
        // 进一步检查Video Renderer之前是不是Overlay Mixer
        IBaseFilter * pUpFilter = GetUpstreamFilter(pVR);
        // 如果Overlay Mixer存在，则定位到再上一级Filter
        if (IsOverlayMixer(pUpFilter))
        {
            pOverlayMixer = pUpFilter;
            pUpFilter = GetUpstreamFilter(pOverlayMixer);
        }
        // 将Video Renderer从Filter Graph中删除
        if (pVR)
        {
            mFilterGraph->GetGraph()->RemoveFilter(pVR);
        }
        // 将Overlay Mixer从Filter Graph中删除
        if (pOverlayMixer)
        {
            mFilterGraph->GetGraph()->RemoveFilter(pOverlayMixer);
        }
        // 得到查询到的上级Filter的输出Pin
        foundPin = GetPin(pUpFilter, FALSE);
    }
    return foundPin;
}
// 在Filter Graph中查找得到支持某个特定接口的Filter
HRESULT CSimplePlayerDlg::FindFilterByInterface(REFIID riid,
    IBaseFilter** ppFilter)
{
    *ppFilter = NULL;
    if (!mFilterGraph)
    {
        return E_FAIL;
    }
    // 得到Filter的枚举器
    IEnumFilters* pEnum;
    HRESULT hr = mFilterGraph->GetGraph()->EnumFilters(&pEnum);
    if (FAILED(hr))
    {
        return hr;
    }

    // 枚举当前Filter Graph中所有的Filter
    IBaseFilter* pFilter = NULL;
    BOOL found = FALSE;
    while (!found && pEnum->Next(1, &pFilter, NULL) == S_OK)
    {
```

```
        // 查询当前得到的Filter是否支持给定的接口
        IUnknown* pUnk;
        HRESULT hrQuery = pFilter->QueryInterface(riid, (void**)&pUnk);
        if (SUCCEEDED(hrQuery))
        {
            pUnk->Release();
            *ppFilter = pFilter;
            found = TRUE;
        }
        pFilter->Release();
    }
    pEnum->Release();
    return (*ppFilter != NULL) ? S_OK : E_FAIL;
}
// 得到指定Filter上的满足一定条件的Pin，匹配条件可以是Pin的方向，或者Pin的名字
IPin * GetPin(IBaseFilter * inFilter, BOOL inInput, const char *
    inPartialName)
{
    if (!inFilter)
    {
        return NULL;
    }
    PIN_DIRECTION direction = inInput ? PINDIR_INPUT : PINDIR_OUTPUT;
    IPin * foundPin = NULL;
    IEnumPins * pinEnum = NULL;
    // 枚举Filter上所有的Pin
    if (SUCCEEDED(inFilter->EnumPins(&pinEnum)))
    {
        pinEnum->Reset();

        IPin * pin = NULL;
        ULONG fetchCount = 0;
        while (!foundPin && SUCCEEDED(pinEnum->Next(1, &pin, &fetchCount))
            && fetchCount)
        {
            if (pin)
            {
                // 得到当前Pin的描述信息
                PIN_INFO pinInfo;
                if (SUCCEEDED(pin->QueryPinInfo(&pinInfo)))
                {
                    // 检查当前Pin的方向
                    if (pinInfo.dir == direction)
                    {
                        // 进一步匹配Pin的名字
                        if (!inPartialName)
                        {
                            pin->AddRef();
                            foundPin = pin;
                        }
```

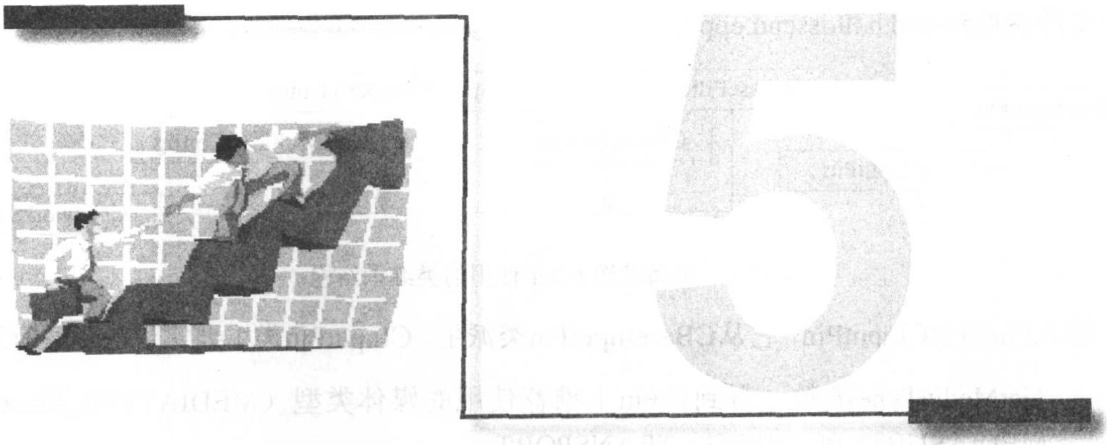
```
                        else
                        {
                            char pinName[128];
                            ::WideCharToMultiByte(CP_ACP, 0, pinInfo.
                                achName, -1, pinName, 128, NULL, NULL);
                            if (::strstr(pinName, inPartialName))
                            {
                                pin->AddRef();
                                foundPin = pin;
                            }
                        }
                    }
                    pinInfo.pFilter->Release();
                }
                pin->Release();
            }
        }
        pinEnum->Release();
    }
    // 本函数不增加返回Pin的引用计数
    if (foundPin)
    {
        foundPin->Release();
    }
    return foundPin;
}
// 得到指定Filter（连接着）的上一级Filter，参数inFilter局限于仅有一个输入Pin的
// Filter
IBaseFilter * GetUpstreamFilter(IBaseFilter* inFilter)
{
    if (!inFilter)
    {
        return NULL;
    }
    // 枚举Filter上所有的Pin
    IBaseFilter * pUpFilter = NULL;
    IEnumPins * pinEnum = NULL;
    if (SUCCEEDED(inFilter->EnumPins(&pinEnum)))
    {
        pinEnum->Reset();

        IPin * pin = NULL;
        ULONG fetchCount = 0;
        while (!pUpFilter && SUCCEEDED(pinEnum->Next(1, &pin, &fetchCount))
            && fetchCount)
        {
            if (pin)
            {
                // 检查当前Pin是否连接
                IPin * connectedPin = NULL;
```

```
                pin->ConnectedTo(&connectedPin);
                if (connectedPin)
                {
                    connectedPin->Release();
                    // 检查连接对方Pin的方向，如果连接对方是输出Pin，则该Pin所属
                    // 的Filter就是我们要找的
                    PIN_INFO pinInfo;
                    if (SUCCEEDED(connectedPin->QueryPinInfo(&pinInfo)))
                    {
                        pinInfo.pFilter->Release();
                        if (pinInfo.dir == PINDIR_OUTPUT)
                        {
                            pUpFilter = pinInfo.pFilter;
                        }
                    }
                }
                pin->Release();
            }
        }
        pinEnum->Release();
    }
    return pUpFilter;
}
// 判断指定的Filter是不是Overlay Mixer
BOOL IsOverlayMixer(IBaseFilter * inFilter)
{
    if (!inFilter)
    {
        return FALSE;
    }
    // 通过Filter上是否支持IAMOverlayFX接口来判断
    IAMOverlayFX * pMixer = NULL;
    inFilter->QueryInterface(IID_IAMOverlayFX, (void**)&pMixer);
    if (pMixer)
    {
        pMixer->Release();
    }
    return (pMixer != NULL);
}
```

播放器程序修改完成之后就可以运行了，如图4.9所示。在播放媒体文件的时候，可以看到一行字符从视频窗口的底部滚过。不过，当在界面上更改了欲叠加的字符内容后，视频图像上叠加的字符内容并不会马上更新，而要将媒体文件通过“打开”按钮重新打开一次才行。为什么？有兴趣的读者可以继续研究一下。

图 4.9　支持字符叠加的媒体文件播放器

提示： 本书配套光盘的SourceCodes\Chapter04\TitleOverlay\SimplePlayer目录下提供了更新版的简单媒体文件播放器的完整实现。



SDK源码赏析

5.1 DSNetwork例子

5.1.1 程序结构分析

DSNetwork是一个DirectShow SDK 9.0新增的Filter例子；它实现了MPEG2传输流的组播（Multicast）发送和接收。（MPEG2定义了两种流格式：传输流（Transport Stream，TS）和程序流（Program Stream，PS）。）这个例子位于SDK安装目录的Samples\C++\ DirectShow\Filters\DSNetwork下。

使用VC 6.0打开工作区文件DSNetwork.dsw，可以看到如下5个项目，它们实现的功能分别是：

- **dsnet**：生成一个DLL（或者说AX文件），实现了Filter的注册函数、反注册函数、以及用于组件对象创建的类工厂模板。
- **dsnetifc**：生成一个静态Lib（被dsnet项目连接），定义了IMulticastConfig接口。
- **dsrecv**：生成一个静态Lib（被dsnet项目连接），实现了一个组播接收的Filter。
- **dssend**：生成一个静态Lib（被dsnet项目连接），实现了一个组播发送的Filter。
- **nutil**：生成一个静态Lib（被dsnet项目连接），实现了几个工具类，如Filter的属性页开发时用到的控件包装类等；还有一些工具函数，如给定一个IP地址，判断它是组播地址还是单播地址等。

DSNetwork实际上实现了两个Filter，它们的注册名字分别是MPEG-2 Multicast Sender和MPEG-2 Multicast Receiver（前者是组播发送，后者是组播接收）。下面就来依次介绍这

两个Filter的具体实现。

组播发送Filter有一个输入Pin，没有输出Pin。Filter和Pin采用的类继承结构如图5.1所示（源文件参见dssend.h和dssend.cpp）。

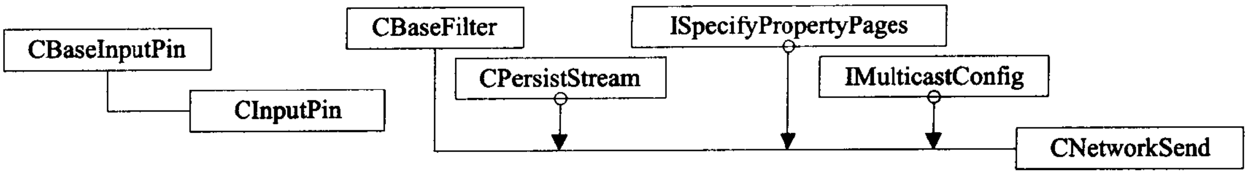


图 5.1　组播发送 Filter 使用的类继承结构

输入Pin类为CInputPin，它从CBaseInputPin类派生。CInputPin类主要实现的函数有3个：

- GetMediaType函数，得到该Pin上推荐使用的媒体类型（MEDIATYPE_Stream，MEDIASUBTYPE_MPEG2_TRANSPORT）。
- CheckMediaType函数，该Pin连接过程中对连接用的媒体类型进行检查，仅接受（MEDIATYPE_Stream，MEDIASUBTYPE_MPEG2_TRANSPORT）的媒体类型。
- Receive函数，上一级Filter通过该函数将数据传入，然后委托给Filter上的Send函数进行实际的处理（Filter的Send函数会调用CNetSender::Send函数完成数据的组播发送）。

Filter类为CNetworkSend，它从CBaseFilter类派生。Filter类实现的框架函数主要有：

- CreateInstance函数，创建一个CNetworkSend类的实例（即Filter组件对象）。
- GetPinCount函数，返回1，表示该Filter仅有一个Pin。
- GetPin函数，返回输入Pin对象指针。
- NonDelegatingQueryInterface函数，暴露Filter上新实现的接口，如IspecifyProperty Pages、IMulticastConfig、IPersistStream等。
- Pause函数，当Filter从停止状态转入暂停状态时，创建一个用于向网络发送数据的Socket，并将其加入到一个组播组（Multicast Group）；实现细节参见CNetSender::JoinMulticast函数。
- Stop函数，当Filter进入停止状态时，销毁Socket资源，也就脱离了组播组；实现细节参见CNetSender::LeaveMulticast函数。

CNetworkSend类除了实现了必要的Filter框架函数外，还实现了一系列接口，如IMulticastConfig接口可以让用户设置组播组的IP地址、本地Socket绑定的端口号、使用的网卡地址等；ISpecifyPropertyPages接口用于指定Filter的属性页；IPersistStream接口用于将Filter内部参数保存到GRF文件（这样的话，使用GraphEdit装载以前构建的Filter Graph文件时，Filter的内部参数可以自动得到恢复）。我们看到，在Filter上实现一个接口，只需将Filter类从这个接口公有派生下来就可以了。这里的IPersistStream接口的实现有点不一样，因为IPersistStream接口已经在CPersistStream类中实现了，CNetworkSend类只需从CPersistStream类再次派生，然后实现CPersistStream类的GetClassID、WriteToStream、ReadFromStream和SizeMax 4个函数即可。

其实，组播功能的真正实现在CNetSender类中（参见源文件netsend.h和netsend.cpp），这个类实现了3个主要的功能函数：JoinMulticast、LeaveMulticast和Send。

```
//
// netsend.cpp
//
// 加入到一个指定的组播组
HRESULT CNetSender::JoinMulticast (
    IN  ULONG   ulIP,
    IN  USHORT  usPort,
    IN  ULONG   ulNIC,
    IN  ULONG   ulTTL
    )
{
    BOOL                t ;
    int                 i ;
    struct sockaddr_in  saddr ;
    DWORD               dw ;

    LeaveMulticast () ;
    ASSERT (m_hSocket == INVALID_SOCKET) ;
    // 创建一个使用UDP协议的Socket
    m_hSocket = WSASocket (
        AF_INET,
        SOCK_DGRAM,
        0,
        NULL,
        0,
        WSA_FLAG_MULTIPOINT_C_LEAF | WSA_FLAG_MULTIPOINT_D_LEAF) ;
    if (m_hSocket == INVALID_SOCKET) {
        goto JoinFail ;
    }
    // 调整Socket上的参数
    t = TRUE ;
    i = setsockopt (
                m_hSocket,
                SOL_SOCKET,
                SO_REUSEADDR,    // 复用地址
                (char *) & t,
                sizeof t
                ) ;
    if (i == SOCKET_ERROR) {
        goto JoinFail ;
    }

    ZeroMemory (& saddr, sizeof saddr) ;
    saddr.sin_family                = AF_INET ;
    saddr.sin_port                  = usPort ;      // 指定的端口号
    saddr.sin_addr.S_un.S_addr      = INADDR_ANY ;  // 不关心本地网卡地址
    // 将Socket绑定到指定的端口上
    i = bind (
            m_hSocket,
```

```
            (LPSOCKADDR) & saddr,
            sizeof saddr
            ) ;
    if (i == SOCKET_ERROR) {
        goto JoinFail ;
    }
    i = setsockopt (
            m_hSocket,
            IPPROTO_IP,
            IP_MULTICAST_TTL,    // Time-To-Live
            (char *) & ulTTL,
            sizeof ulTTL
            ) ;
    if (i == SOCKET_ERROR) {
        goto JoinFail ;
    }
    i = setsockopt (
            m_hSocket,
            IPPROTO_IP,
            IP_MULTICAST_IF,
            (char *) & ulNIC,
            sizeof ulNIC
            ) ;
    if (i == SOCKET_ERROR) {
        goto JoinFail ;
    }

    ZeroMemory (& m_saddrDest, sizeof m_saddrDest) ;
    m_saddrDest.sin_family              = AF_INET ;
    m_saddrDest.sin_port                = usPort ;
    m_saddrDest.sin_addr.S_un.S_addr    = ulIP ;
    return TRUE ;

    JoinFail:
    dw = WSAGetLastError () ;  // 得到Socket错误码
    if (dw == NOERROR) {
        dw = ERROR_GEN_FAILURE ;
    }
    LeaveMulticast () ;
    return HRESULT_FROM_WIN32 (dw) ;
}
// 脱离组播组
HRESULT CNetSender::LeaveMulticast ( )
{
    // 释放Socket资源
    if (m_hSocket != INVALID_SOCKET) {
        closesocket (m_hSocket) ;
        m_hSocket = INVALID_SOCKET ;
    }
    return S_OK ;
}
// 组播方式发送数据
```

```
HRESULT CNetSender::Send (
    IN BYTE * pbBuffer,
    IN DWORD  dwLength
    )
{
    HRESULT hr ;
    DWORD   dw ;
    DWORD   dwSnarf ;
    int     i ;

    ASSERT (pbBuffer) ;

    if (m_hSocket != INVALID_SOCKET) {
        while (dwLength > 0) {
            dwSnarf = Min <DWORD> (m_dwPTTransmitLength, dwLength) ;
            // 发送数据（每次发送的数据量不超过m_dwPTTransmitLength)
            i = sendto (
                    m_hSocket,
                    (const char *) pbBuffer,
                    dwSnarf,
                    0,
                    (LPSOCKADDR) & m_saddrDest,
                    sizeof m_saddrDest
                    ) ;
            if (i != (int) dwSnarf) {
                dw = GetLastError () ;
                return HRESULT_FROM_WIN32 (dw) ;
            }
            pbBuffer += dwSnarf ;
            dwLength -= dwSnarf ;
        }
        hr = S_OK ;
    }
    else {
        hr = E_UNEXPECTED ;
    }
    return hr ;
}
```

提示：CNetSender类在构造函数中调用了WSAStartup函数进行Socket库的初始化，在析构函数中调用了WSACleanup函数进行Socket库的反初始化。这一对函数的调用，对于正常使用Socket功能是非常重要的。

最后来看一下组播发送Filter的属性页的实现（参见源文件propsend.h和propsend.cpp）。这个属性页实际上演示了IMulticastConfig接口的使用方法；用户通过该属性页界面，可以设置组播组的IP地址、端口号、网卡地址等一系列参数。属性页的实现很简单，首先在VC的资源编辑器中生成一个对话框页面，并且添加必要的控件；然后自定义一个CNetSendProp类从CBasePropertyPage类派生，并且重写CBasePropertyPage类的一些函数即可。值得注意的是，属性页也是一个COM组件，因此要实现组件对象的创建函数CreateInstance，并且要

在类工厂模板中定义。CNetSendProp类的具体实现这里就不做展开论述了。

提示：dssend项目中其他一些源文件的作用：main.cpp，定义组播发送Filter的注册信息；projpch.h，DSNetwork的几个项目公共的头文件，用于定义一些宏、模板函数、字符串，包含公共的头文件，如streams.h、winsock2.h等；precomp.h、precomp.cpp，dssend项目的预编译源文件。

下面来看组播接收Filter。组播接收Filter的实现要比组播发送Filter复杂一点，这个Filter有一个输出Pin，没有输入Pin。Filter和Pin采用的类继承结构如图5.2所示（源文件参见dsrecv.h和dsrecv.cpp）。

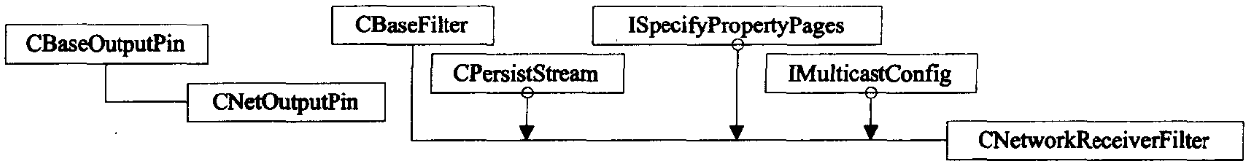


图 5.2　组播接收 Filter 使用的类继承结构

输出Pin类为CNetOutputPin，它从CBaseOutputPin类派生。CNetOutputPin类主要实现的函数有4个：

- GetMediaType函数，得到该Pin上推荐使用的媒体类型（MEDIATYPE_Stream，MEDIASUBTYPE_MPEG2_TRANSPORT）。
- CheckMediaType函数，该Pin连接过程中对连接用的媒体类型进行检查，仅接受（MEDIATYPE_Stream，MEDIASUBTYPE_MPEG2_TRANSPORT）的媒体类型。
- DecideBufferSize函数，决定该Pin使用的Sample管理器的属性，如Sample内存的大小、Sample的个数等。
- InitAllocator函数，创建一个该Pin使用的Sample管理器对象（实际由Filter对象来创建，这里只是获取Sample管理器对象的指针）。

Filter类为CNetworkReceiverFilter，它从CBaseFilter类派生。Filter类实现的框架函数主要有：

- CreateInstance函数，创建一个CNetworkReceiverFilter类的实例（即Filter组件对象）。
- GetPinCount函数，返回1，表示该Filter仅有一个Pin。
- GetPin函数，返回输出Pin对象指针。
- NonDelegatingQueryInterface函数，暴露Filter上新实现的接口，如IspecifyPropertyPages、IMulticastConfig、IPersistStream等。
- Pause函数，当Filter从停止状态转入暂停状态时，创建一个用于接收网络数据的Socket，并将其加入到一个组播组（Multicast Group），然后启动数据接收线程，实现细节参见CNetReceiver::Activate函数。
- Stop函数，当Filter进入停止状态时，销毁Socket资源，也就脱离了组播组，然后停止数据接收线程，实现细节参见CNetReceiver::Stop函数。

组播接收Filter上实现的接口与组播发送Filter很类似，如IMulticastConfig接口让用户设置组播组的IP地址、端口号、本地的网卡地址等；ISpecifyPropertyPages接口用于指定Filter的属性页；IPersistStream接口用于将Filter内部参数保存到GRF文件（这样的话，使用GraphEdit装载以前构建的Filter Graph文件时，Filter的内部参数可以自动得到恢复）。

此外，还有一个很重要的问题：组播接收Filter是一个推模式的Source Filter，那么它的“推”数据功能是如何来实现的呢？带着这个问题，我们来看dsrecv项目实现的一系列的工具类（其中几个工具类的继承结构见图5.3）：

- 自定义缓存类CBuffer（参见源文件buffpool.h和buffpool.cpp），封装了一定大小的内存（dsrecv项目中定义MAX_IOBUFFER_LENGTH为8192字节），用于一块数据的读写，可以放到缓存池中循环使用。
- 缓存池CBufferPool（参见源文件buffpool.h和buffpool.cpp），维护了一个包含一定数量的（dsrecv项目中定义POOL_SIZE为24）CBuffer类型的缓存对象的双向列表。这个列表中的每一个缓存对象都能被重用：外部调用者需要用缓存时，从列表中取出一个空闲的缓存对象，然后就是使用；使用完之后，将该缓存对象回收到列表中，等待下一次的使用。
- 自定义Sample类CTSMediaSample（参见源文件mspool.h和mspool.cpp）。因为DirectShow中以Sample的形式来传递数据，所以要定义这个类来包装CBuffer类型的缓存对象。
- Sample池CTSMediaSamplePool（参见源文件mspool.h和mspool.cpp），维护了一个包含一定数量的（dsrecv项目中定义MEDIA_SAMPLE_POOL_SIZE为10）CTSMediaSample类型的Sample对象的双向列表。这个列表中的每一个Sample对象也都是可以被重用的。
- 自定义Sample管理器CNetRecvAlloc（参见源文件alloc.h和alloc.cpp）。DirectShow的标准设计中，使用Sample管理器（Allocator）来管理Pin上使用的Sample。既然dsrecv项目中已经使用CTSMediaSamplePool实现了类似的功能，CNetRecvAlloc类就不再需要重复实现了。因此，CNetRecvAlloc类实现的一些函数，要么直接返回S_OK，要么直接返回E_NOTIMPL。

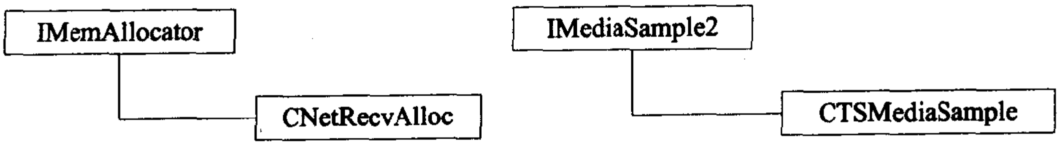


图5.3 组播接收Filter用到的几个工具类的继承结构

控制类CNetReceiver（参见源文件netrecv.h和netrecv.cpp）正是使用上述工具类，实现了组播数据的（异步方式）接收，然后以Sample的形式“推”给下一级Filter的整个控制流程。CNetReceiver类中封装了一个控制线程。当外部调用者（Filter对象）调用Activate函数时，创建一个接收网络数据用的Socket，并将其加入到指定的组播组中，然后就是启动这个控制线程，开始一个从网络数据接收到Sample发送的反复过程。

```
//
// netrecv.cpp
//
// 创建接收网络数据用的Socket并将其加入到组播组，启动控制线程
// 注意：参数均采用网络字节顺序，即big-endian
HRESULT CNetReceiver::Activate (
    IN  ULONG   ulIP,           //  IP地址
    IN  USHORT  usPort,         //  端口号
    IN  ULONG   ulNIC           //  网卡
    )
{
    HRESULT hr ;
    DWORD   dw ;

    Lock_ () ;
    ASSERT (IsMulticastIP (ulIP)) ;
    ASSERT (m_hThread == NULL) ;
    ASSERT (m_hEvents [EVENT_STOP] != NULL) ;
    // 初始化事件状态
    ResetEvent (m_hEvents [EVENT_STOP]) ;
    m_lReadsPended = 0 ;
    // 创建接收网络数据用的Socket并将其加入到组播组，启动控制线程
    hr = JoinMulticast_ (ulIP, usPort, ulNIC) ;
    if (SUCCEEDED (hr)) {
        // 创建一个控制线程
        m_hThread = CreateThread (
                        NULL,
                        0,
                        (LPTHREAD_START_ROUTINE) CNetReceiver::ThreadEntry,
                        this,
                        NULL,
                        & dw
                        ) ;
        if (m_hThread) {
            hr = S_OK ;
            ASSERT (m_hAsyncSocket != INVALID_SOCKET) ;
        }
        else {
            dw = GetLastError () ;
            LeaveMulticast_ () ;
            hr = HRESULT_FROM_WIN32 (dw) ;
        }
    }
    else {
        ASSERT (m_hAsyncSocket == INVALID_SOCKET) ;
    }

    Unlock_ () ;
    return S_OK ;
}
// 控制线程的函数执行体
static DWORD WINAPI ThreadEntry (IN  LPVOID  pv)
```

```
{
    (reinterpret_cast <CNetReceiver *> (pv)) -> ThreadProc () ;
    return EXIT_SUCCESS ;
}
// 控制线程中网络数据接收、Sample发送的循环过程
void CNetReceiver::ThreadProc ( )
{
    DWORD   r ;
    for (;;) {
        // 阻塞等待事件的有效状态（等待时间PULSE_MILLIS定义为100ms）
        r = WaitForMultipleObjectsEx (EVENT_COUNT, m_hEvents, FALSE,
            PULSE_MILLIS, TRUE) ;
        // 如果停止事件有效，则跳出这个死循环
        if (r - WAIT_OBJECT_0 == EVENT_STOP) {
            break ;
        }
        // 如果等待超时，则发出一个接收数据的请求
        if (r == WAIT_TIMEOUT) {
            PendReads_ () ;
        }
    }
    return ;
}
// 发出一个接收网络数据的异步请求
void CNetReceiver::PendReads_ (
    IN  DWORD   dwBufferWaitMax
    )
{
    CBuffer *   pBuffer ;
    int         i ;
    WSABUF      wsabuf ;
    DWORD       dwBytesRead ;
    DWORD       dw ;
    DWORD       dwFlags ;

    Lock_ () ;
    if (m_hAsyncSocket != INVALID_SOCKET) {

        // 不要发出太多的数据接收请求（不超过10个）
        // 如果未完成的请求超过10个，则等待这些请求完成一部分后再发出新的请求
        // m_lReadsPended记录了当前没有完成的请求总数
        while (m_lReadsPended < MAX_READ_PENDS) {
            // 从缓存池中取出一个空闲的缓存
            pBuffer = m_pBufferPool -> GetBuffer (
                m_hEvents [EVENT_GET_BLOCK],
                dwBufferWaitMax
                ) ;
            if (pBuffer) {
                pBuffer -> GetOverlapped () -> Offset       = 0 ;
                pBuffer -> GetOverlapped () -> OffsetHigh   = 0 ;
                pBuffer -> GetOverlapped () -> hEvent       = (HANDLE) pBuffer ;
                pBuffer -> SetCompletionContext ((DWORD_PTR) this) ;
```

```
                wsabuf.buf  = reinterpret_cast <char *> (pBuffer -> GetBuffer
                    ()) ;
                wsabuf.len  = pBuffer -> GetBufferLength () ;
                dwFlags = 0 ;
                // 发出一个异步接收数据的请求
                // 当一个异步请求完成时会调用AsyncCompletionCallback函数
                i = WSARecv (
                        m_hAsyncSocket,
                        & wsabuf,
                        1,
                        & dwBytesRead,
                        & dwFlags,
                        pBuffer -> GetOverlapped (),
                        CNetReceiver::AsyncCompletionCallback
                        ) ;
                if (i == SOCKET_ERROR) {
                    dw = WSAGetLastError () ;
                    if (dw != WSA_IO_PENDING) {
                        pBuffer -> Release () ;
                        break ;
                    }
                }
                // 增加一个未完成请求的计数
                InterlockedIncrement (& m_lReadsPended) ;
            }
            else {
                break ;
            }
        }
    }
    Unlock_ () ;
}
// 完成网络数据异步接收请求后调用的回调函数
void CALLBACK CNetReceiver::AsyncCompletionCallback (
    IN  DWORD            dwError,
    IN  DWORD            dwBytesReceived,
    IN  LPWSAOVERLAPPED  pOverlapped,
    IN  DWORD            dwFlags
    )
{
    CBuffer *       pBuffer ;
    CNetReceiver *  pNetReceiver ;
    // 得到这次请求操作的缓存对象
    pBuffer = reinterpret_cast <CBuffer *> (pOverlapped -> hEvent) ;
    // 得到控制对象（就是CNetReceiver类的实例）指针
    pNetReceiver = reinterpret_cast <CNetReceiver *> (pBuffer ->
        GetCompletionContext ()) ;
    // 设置实际接收到的网络数据长度
    pBuffer -> SetPayloadLength (dwBytesReceived) ;
    // 最后调用控制对象上的ReadCompletion函数
    pNetReceiver -> ReadCompletion (pBuffer, dwError) ;
}
```

```
// 一个网络数据接收请求完成后最后调用的函数：将接收到的数据以Sample的形式“推”给下一
// 级Filter
void CNetReceiver::ReadCompletion (
    IN  CBuffer *   pBuffer,
    IN  DWORD       dwError
    )
{
    // 将未完成的接收请求计数减1
    InterlockedDecrement (& m_lReadsPended) ;
    // 发出一个新的网络数据接收请求
    PendReads_ () ;
    if (dwError == NO_ERROR) {
        // 将数据交给Filter对象处理，Filter对象将以Sample的形式“推”给下一级
        // Filter
        m_pRecvFilter -> ProcessBuffer (pBuffer) ;
}
// 回收当前完成一次使用的缓存对象（以备下次循环使用）
pBuffer -> Release () ;
}

//
// dsrecv.cpp
//
// 将接收到的网络数据以Sample的形式“推”给下一级Filter
void CNetworkReceiverFilter::ProcessBuffer (
    IN  CBuffer *   pBuffer
    )
{
    HRESULT         hr ;
    IMediaSample2 * pIMediaSample ;
    ASSERT (pBuffer) ;
    // 从Sample池中取出一个空闲的Sample,并用这个Sample来包装进来的CBuffer类型的缓
    // 存对象
    hr = m_pMSPool -> GetMediaSampleSynchronous (
          pBuffer,
          pBuffer -> GetBuffer (),
          pBuffer -> GetPayloadLength (),
          & pIMediaSample
          ) ;
    if (SUCCEEDED (hr)) {
        ASSERT (pIMediaSample) ;
        // 将Sample传送下去
        m_pOutput -> Deliver (pIMediaSample) ;
        // 回收使用过的Sample（以备下次循环使用）
        pIMediaSample -> Release () ;
    }
}
```

当组播接收Filtert停止运行时，Filter对象会调用CNetReceiver::Stop函数，将上述这个网络数据接收、Sample发送的过程终止。

```
//
// netrecv.cpp
//
HRESULT CNetReceiver::Stop ( )
{
    // 将停止事件设置为有效（让控制线程退出）
    SetEvent (m_hEvents [EVENT_STOP]) ;
    // 销毁Socket资源，脱离组播组
    LeaveMulticast_ () ;
    if (m_hThread) {
        // 等待控制线程的完全退出
        WaitForSingleObject (m_hThread, INFINITE) ;
        CloseHandle (m_hThread) ;
        m_hThread = NULL ;
    }
    return S_OK ;
}
```

提示：dsrecv项目中其他一些源文件的作用：main.cpp，定义组播接收Filter的注册信息；le.h，实现双向列表元素的一系列操作方法，如IsListEmpty、InsertHeadList、RemoveHeadList等；proprecv.h、proprecv.cpp，实现了组播接收Filter的属性页；projpch.h，DSNetwork的几个项目公共的头文件，用于定义一些宏、模板函数、字符串，包含公共的头文件，如streams.h、winsock2.h等；precomp.h、precomp.cpp，dsrecv项目的预编译源文件。

5.1.2 模拟MPEG2-TS源

DSNetwork例子实现了MPEG2 TS流的组播发送和接收。那么，如何来使用这两个Filter呢？

首先要开发一个模拟发送MPEG2 TS流的Source Filter。这个Source Filter要求工作在推模式下，注册名为MPEG2 TS Source，而且仅需要一个输出Pin，如图5.4所示。

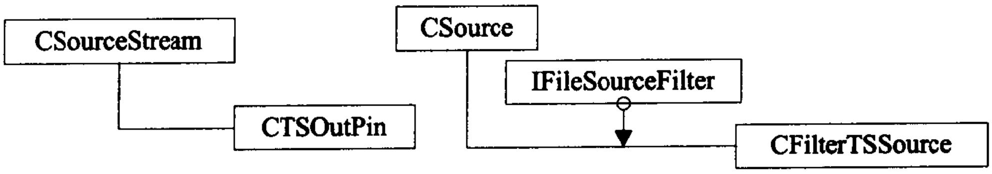


图 5.4 MPEG2 TS Source Filter 使用的类继承结构

另外，为了给MPEG2 TS Source Filter指定一个包含MPEG2 TS流数据的源文件，也为了在GraphEdit中插入该Filter时自动弹出一个文件浏览对话框让用户指定源文件，MPEG2 TS Source Filter上需要实现一个IFileSourceFilter接口。

```
//
// CTSOutPin.h
//

#ifndef __H_CTSOutPin__
```

```
#define __H_CTSOutPin__

#define SAMPLE_SIZE 1880    // Sample的大小

#include <stdio.h>

class CFilterTSSource;
class CTSOutPin : public CSourceStream
{
    friend class CFilterTSSource;
protected:
    CFilterTSSource *   mFilter;              // Filter对象指针
    CCritSec            mSharedState;
    char                mFilePath[MAX_PATH];  // 源文件路径
    FILE *              m_fp;                 // 文件操作指针
    CMediaType          mPreferred;           // Pin上推荐使用的媒体类型
    LONGLONG            mSampleCount;         // 发出去的Sample计数
public:
    // 设置/得到源文件路径
    BOOL SetFileSource(const char * inFile);
    void GetFileSource(char * outFile);
public:
    CTSOutPin(HRESULT *phr, CFilterTSSource * pFilter, LPCWSTR pPinName);
        ~CTSOutPin();
    // 得到Pin的ID
    STDMETHODIMP QueryId(LPWSTR * Id);
    // 实现纯虚函数
    // 填写Sample的内容
    virtual HRESULT FillBuffer(IMediaSample * pSample);
    // 决定Sample的属性，包括使用的Sample数、Sample大小等
    virtual HRESULT DecideBufferSize(IMemAllocator *pAlloc,
        ALLOCATOR_PROPERTIES *pProperties);
    // Pin连接过程中进行媒体类型的检查
    virtual HRESULT CheckMediaType(const CMediaType * inMediatype);
    // Pin激活（在Filter开始运行时被调用）
    virtual HRESULT Active(void);
    // 得到该Pin上推荐使用的媒体类型
    HRESULT GetMediaType(int iPosition, CMediaType *pmt);
    // 质量控制（Quality Control）处理
    STDMETHODIMP Notify(IBaseFilter * pSender, Quality q);
    // 推数据线程开始运行时进行一些初始化
    virtual HRESULT OnThreadStartPlay(void);
};

#endif // __H_CTSOutPin__

//
// CTSOutPin.cpp
//

#include <streams.h>
#include "CFilterTSSource.h"
```

```
#include "CTSOutPin.h"

//////////////////////////////////////////////////////////////////////////////
CTSOutPin::CTSOutPin(HRESULT *phr, CFilterTSSource * pFilter, LPCWSTR
    pPinName) :
CSourceStream(NAME("MPEG2 TS Stream"), phr, pFilter, pPinName)
{
    mFilter = pFilter;
    m_fp    = NULL;
    strcpy(mFilePath, "");
    // 该Pin上推荐使用的媒体类型 (MEDIATYPE_Stream,
    // MEDIASUBTYPE_MPEG2_TRANSPORT)
    mPreferred.InitMediaType();
    mPreferred.SetType(&MEDIATYPE_Stream);
    mPreferred.SetSubtype(&MEDIASUBTYPE_MPEG2_TRANSPORT);
    mSampleCount = 0;
}
CTSOutPin::~CTSOutPin()
{
    // 在Filter销毁之前关闭源文件
    CAutoLock lck(&mSharedState);
    if (m_fp)
    {
        fclose(m_fp);
        m_fp = NULL;
    }
}
// 得到Pin的ID
STDMETHODIMP CTSOutPin::QueryId(LPWSTR * Id)
{
    return CBaseOutputPin::QueryId(Id);
}
// 填写Sample的内容
HRESULT CTSOutPin::FillBuffer(IMediaSample * pSample)
{
    CAutoLock lck(&mSharedState);
    // 得到Sample的内存指针
    BYTE * pData = NULL;
    pSample->GetPointer(&pData);
    // 从源文件的当前位置读取一定数量的数据到Sample内存中
    long bytes = fread(pData, 1, SAMPLE_SIZE, m_fp);
    if (bytes < SAMPLE_SIZE && feof(m_fp))
    {
        // 如果已经到达了文件末尾，则返回S_FALSE以结束线程的推数据过程
        // 并且弹出一个消息框提示
        ::MessageBox(NULL, "End Of Stream!", "DSNetwork", MB_OK);
        return S_FALSE;
    }
    // 设置Sample上的有效数据长度
    pSample->SetActualDataLength(bytes);
    pSample->SetSyncPoint(FALSE);
    // 给Sample打上索引号（可选）
```

```
    LONGLONG start = mSampleCount++;
    LONGLONG stop  = mSampleCount;
    pSample->SetMediaTime(&start, &stop);
    return S_OK;
}
// 决定Sample的属性，包括使用的Sample数、Sample大小等
HRESULT CTSOutPin::DecideBufferSize(IMemAllocator *pAlloc,
    ALLOCATOR_PROPERTIES *pProperties)
{
    CAutoLock  lck(m_pFilter->pStateLock());
    ASSERT(pAlloc);
    ASSERT(pProperties);
    HRESULT hr = NOERROR;
    pProperties->cBuffers = 1;             // 只需使用一个Sample
    pProperties->cbBuffer = SAMPLE_SIZE;   // 每个Sample为1880字节

    ASSERT(pProperties->cbBuffer);
    ALLOCATOR_PROPERTIES Actual;
    hr = pAlloc->SetProperties(pProperties, &Actual);
    if (FAILED(hr))
    {
        return hr;
    }
    if (Actual.cbBuffer < pProperties->cbBuffer)
    {
        return E_FAIL;
    }
    ASSERT( Actual.cBuffers == 1 );
    return NOERROR;
}
// Pin激活时将文件操作指针定位到文件开头
HRESULT CTSOutPin::Active(void)
{
    if (m_fp)
    {
        CAutoLock  lck(&mSharedState);
        fseek(m_fp, 0, SEEK_SET);
    }
    return CSourceStream::Active();
}
// Pin连接过程中进行媒体类型的检查（仅接受MPEG2 TS流）
HRESULT CTSOutPin::CheckMediaType(const CMediaType * inMediatype)
{
    CAutoLock  lck(m_pFilter->pStateLock());
    if (inMediatype->majortype == MEDIATYPE_Stream &&
        inMediatype->subtype == MEDIASUBTYPE_MPEG2_TRANSPORT)
    {
        return NOERROR;
    }
    return E_FAIL;
}
// 得到该Pin上推荐使用的媒体类型
```

```
HRESULT CTSOutPin::GetMediaType(int iPosition, CMediaType *pmt)
{
    CAutoLock lck(m_pFilter->pStateLock());
    if (iPosition != 0)
    {
        return E_FAIL;
    }
    *pmt = mPreferred;
    return NOERROR;
}
// 质量控制（Quality Control）处理
STDMETHODIMP CTSOutPin::Notify(IBaseFilter * pSender, Quality q)
{
    return NOERROR;
}
// 推数据线程开始运行时将送出的Sample计数复位
HRESULT CTSOutPin::OnThreadStartPlay(void)
{
    mSampleCount = 0;
    return NOERROR;
}
// 设置源文件
BOOL CTSOutPin::SetFileSource(const char * inFile)
{
    CAutoLock lck(&mSharedState);
    strcpy(mFilePath, inFile);  // 保存文件路径
    // 将以前打开的源文件关闭
    if (m_fp)
    {
        fclose(m_fp);
        m_fp = NULL;
    }
    // 将源文件打开，准备读取数据
    m_fp = fopen(mFilePath, "rb");
    return (m_fp != NULL);
}
// 得到当前使用的源文件
void CTSOutPin::GetFileSource(char * outFile)
{
    strcpy(outFile, mFilePath);
}

//
// CFilterTSSource.h
//

#ifndef __H_CFilterTSSource__
#define __H_CFilterTSSource__

#include "CTSOutPin.h"

class CFilterTSSource : public CSource, public IFileSourceFilter
```

```
{
private:
    CFilterTSSource(LPUNKNOWN lpunk, HRESULT *phr);
    ~CFilterTSSource();
public:
    static CUnknown * WINAPI CreateInstance(LPUNKNOWN lpunk, HRESULT *phr);
    DECLARE_IUNKNOWN;
    // 用以暴露IFileSourceFilter接口
    STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void ** ppv);
    virtual int GetPinCount();
    virtual CBasePin *GetPin(int n);
    STDMETHODIMP FindPin(LPCWSTR Id, IPin ** ppPin);
    // --- IFileSourceFilter接口函数 ---
    STDMETHODIMP Load(LPCOLESTR pszFileName, const AM_MEDIA_TYPE *pmt);
    STDMETHODIMP GetCurFile(LPOLESTR *ppszFileName, AM_MEDIA_TYPE *pmt);

protected:
    CTSOutPin * OutPin() {return (CTSOutPin *)m_paStreams[0];};
};

#endif // __H_CFilterTSSource__

//
// CFilterTSSource.cpp
//

#include <streams.h>
#pragma warning(disable: 4511 4512)
#include <initguid.h>
// Filter的CLSID定义
// {237204D5-8343-400e-8037-B3C20DB2AB22}
DEFINE_GUID(CLSID_MPEG2_TS_Source, 0x237204d5,
    0x8343, 0x400e, 0x80, 0x37, 0xb3, 0xc2, 0xd, 0xb2, 0xab, 0x22);

#include "CFilterTSSource.h"

// 以下是Filter的注册信息填写
const AMOVIESETUP_MEDIATYPE sudPinTypes =
{
   &MEDIATYPE_NULL,              // 主媒体类型
   &MEDIASUBTYPE_NULL            // 辅助说明类型
};
const AMOVIESETUP_PIN psudPins[] =
{
  {
        L"Output",            // Pin的名字
        FALSE,                // 是最终的Render Pin吗?
        TRUE,                 // 是输出Pin吗?
        FALSE,                // 允许没有这个Pin?
        FALSE,                // 允许有多个这样的Pin?
        &CLSID_NULL,          // 指定连接的Filter
        L"Input",             // 指定连接的Pin
```

```
        1,                  // 支持的媒体类型个数
        &sudPinTypes        // Pin上支持的媒体类型
    }
};
const AMOVIESETUP_FILTER sudFilter =
{
    &CLSID_MPEG2_TS_Source,      // Filter的CLSID
    L"MPEG2 TS Source",          // Filter的名字
    MERIT_DO_NOT_USE,            // Filter的Merit值
    1,                           // Pin的个数
    psudPins                     // Pin的细节描述
};
CFactoryTemplate g_Templates[] =
{
    {
        L"MPEG2 TS Source",                  // Filter的注册名
        &CLSID_MPEG2_TS_Source,              // Filter的CLSID
        CFilterTSSource::CreateInstance,     // 创建Filter的入口函数
        NULL,         // 初始化函数地址
        &sudFilter  // Filter的描述信息
    }
};
int g_cTemplates = sizeof(g_Templates) / sizeof(g_Templates[0]);

// ----------------------------------------------------------------------
//           Filter implementation
// ----------------------------------------------------------------------
CFilterTSSource::CFilterTSSource(LPUNKNOWN lpunk, HRESULT *phr) :
CSource(NAME("TS Source"), lpunk, CLSID_MPEG2_TS_Source)
{
    // 创建Pin的实例
    CTSOutPin * outStream = new CTSOutPin(phr, this, L"Output");
    if (outStream == NULL)
    {
        *phr = E_OUTOFMEMORY;
    }
}
CFilterTSSource::~CFilterTSSource()
{
}
// 创建Filter对象
CUnknown * WINAPI CFilterTSSource::CreateInstance(LPUNKNOWN punk, HRESULT
    *phr)
{
    CFilterTSSource *pNewObject = new CFilterTSSource(punk, phr);
    if (pNewObject == NULL)
    {
        *phr = E_OUTOFMEMORY;
    }
    return pNewObject;
}
// 暴露本Filter上实现的接口
```

```
STDMETHODIMP CFilterTSSource::NonDelegatingQueryInterface(REFIID riid,
    void ** ppv)
{
    CheckPointer(ppv, E_POINTER);

    if (riid == IID_IFileSourceFilter)
    {
        return GetInterface((IFileSourceFilter *) this, ppv);
    }
    else
    {
        return CSource::NonDelegatingQueryInterface(riid, ppv);
    }
}
// 得到Filter拥有Pin的个数
int CFilterTSSource::GetPinCount()
{
    return 1;
}
// 得到各个Pin对象指针
CBasePin * CFilterTSSource::GetPin(int n)
{
    if (n == 0)
    {
        return m_paStreams[0];
    }
    return NULL;
}
// 根据Pin的ID找到对应的Pin
STDMETHODIMP CFilterTSSource::FindPin(LPCWSTR Id, IPin ** ppPin)
{
    return CBaseFilter::FindPin(Id, ppPin);
}
// --- IFileSourceFilter的接口函数实现---
// 装载一个源文件
STDMETHODIMP CFilterTSSource::Load(LPCOLESTR pszFileName, const
    AM_MEDIA_TYPE *pmt)
{
    // 将宽字符转换成ANSI字符
    char szAnsi[MAX_PATH];
    WideCharToMultiByte(CP_ACP, 0, pszFileName, -1, szAnsi, MAX_PATH, NULL,
        NULL);
    if (OutPin()->SetFileSource(szAnsi))
    {
        return NOERROR;
    }
    return E_FAIL;
}
// 得到当前使用的源文件
STDMETHODIMP CFilterTSSource::GetCurFile(LPOLESTR *ppszFileName,
    AM_MEDIA_TYPE *pmt)
{
```

```
    // 从Pin对象上得到当前使用的源文件
    char szAnsi[MAX_PATH];
    OutPin()->GetFileSource(szAnsi);
    DWORD n = sizeof(WCHAR) * (1 + strlen(szAnsi));
    *ppszFileName = (LPOLESTR) CoTaskMemAlloc( n );
    if (*ppszFileName != NULL)
    {
        // 将ANSI字符转换成宽字符
        WCHAR   szwFile[MAX_PATH];
        MultiByteToWideChar(CP_ACP, 0, szAnsi, -1, szwFile, MAX_PATH);
        CopyMemory(*ppszFileName, szwFile, n);
    }
    return NOERROR;
}

// DLL入口函数
extern "C" BOOL WINAPI DllEntryPoint(HINSTANCE, ULONG, LPVOID);
BOOL APIENTRY DllMain(HANDLE hModule, DWORD  dwReason, LPVOID lpReserved)
{
    return DllEntryPoint((HINSTANCE)(hModule), dwReason, lpReserved);
}
// 注册函数
STDAPI DllRegisterServer()
{
    return AMovieDllRegisterServer2( TRUE );
}
// 反注册函数
STDAPI DllUnregisterServer()
{
    return AMovieDllRegisterServer2( FALSE );
}
#pragma warning( disable:4514)
```

提示：本书配套光盘的SourceCodes\Chapter05\Multicast\FilterTSSource目录下有MPEG2 TS Source Filter的完整实现。

5.1.3 应用演示

MPEG2 TS Source Filter开发完成并且成功注册后，就可以开始DSNetwork例子的组播发送、接收的演示了。由于在接收端构建MPEG2 TS流的解码链路比较复杂，本演示仅将接收到的数据存储为一个本地文件。因此，需要对组播接收Filter进行如下调整。

```
//
// dsrecv.h
//
// 在Filter类中增加如下成员变量
class CNetworkReceiverFilter
{
#define _DUMP_RECEIVED

#ifdef _DUMP_RECEIVED
```

```
    FILE * m_fp;             // 文件操作指针
    static int sInstances;   // 本Filter的实例索引（本Filter可能被创建多个实例）
    char   mFileName[256];   // 文件名
#endif
};

//
// dsrecv.cpp
//
#ifdef _DUMP_RECEIVED
int CNetworkReceiverFilter::sInstances = 0;
#endif
// Filter类的构造函数
CNetworkReceiverFilter::CNetworkReceiverFilter (
    IN  TCHAR *     tszName,
    IN  LPUNKNOWN   punk,
    OUT HRESULT *   phr
    )
{
    // 创建文件
#ifdef _DUMP_RECEIVED
    m_fp = NULL;
    sInstances++;
    sprintf(mFileName, "C:\\Multicast_Recv_%d.mpg", sInstances);
    m_fp = fopen(mFileName, "wb");
#endif
}
// Filter类的析构函数
CNetworkReceiverFilter::~CNetworkReceiverFilter ()
{
    // 关闭文件
#ifdef _DUMP_RECEIVED
    if (m_fp)
    {
        fclose(m_fp);
        m_fp = NULL;
    }
#endif
}
// Filter上将接收到的网络数据以Sample的形式推给下一级Filter的函数
void CNetworkReceiverFilter::ProcessBuffer (
    IN CBuffer *   pBuffer
    )
{
    HRESULT         hr ;
    IMediaSample2 * pIMediaSample ;
    ASSERT (pBuffer) ;
    hr = m_pMSPool -> GetMediaSampleSynchronous (
         pBuffer,
         pBuffer -> GetBuffer (),
         pBuffer -> GetPayloadLength (),
         & pIMediaSample
```

```
        ) ;
    if (SUCCEEDED (hr)) {
        ASSERT (pIMediaSample) ;
         // 如果定义了_DUMP_RECEIVED宏，则将数据写到文件中保存，否则，按原来的设计
         // 将Sample推下去
#ifdef _DUMP_RECEIVED
    PBYTE pData;
    pIMediaSample->GetPointer(&pData);
    if (m_fp)
    {
        fwrite(pData, 1, pIMediaSample->GetActualDataLength(), m_fp);
    }
#else
        m_pOutput -> Deliver (pIMediaSample) ;
#endif
        pIMediaSample -> Release () ;
    }
}
```

为了验证MPEG2 TS Source Filter是原封不动地将源文件数据发送出来的，在MPEG2 TS Source Filter内也可以做如下调整：把Source Filter将要发送出去的Sample数据写到一个本地文件中去。

```
//
// CTSOutPin.h
//
// 在输出Pin类中增加如下成员变量
class CTSOutPin : public CSourceStream
{
#define _DUMP_SENT

#ifdef _DUMP_SENT
    FILE * m_fpDebug;
#endif
};

//
// CTSOutPin.cpp
//
// 输出Pin类的构造函数
CTSOutPin::CTSOutPin(HRESULT *phr, CFilterTSSource * pFilter, LPCWSTR
    pPinName) :
CSourceStream(NAME("MPEG2 TS Stream"), phr, pFilter, pPinName)
{
#ifdef _DUMP_SENT
    m_fpDebug = fopen("C:\\MediaSent.mpg", "wb");
#endif
}
// 输出Pin类的析构函数
CTSOutPin::~CTSOutPin()
{
#ifdef _DUMP_SENT
```

```
    if (m_fpDebug)
    {
        fclose(m_fpDebug);
        m_fpDebug = NULL;
    }
#endif
}
// 填写Sample的内容
HRESULT CTSOutPin::FillBuffer(IMediaSample * pSample)
{
    CAutoLock  lck(&mSharedState);

    BYTE * pData = NULL;
    pSample->GetPointer(&pData);
    long bytes = fread(pData, 1, SAMPLE_SIZE, m_fp);
    if (bytes < SAMPLE_SIZE && feof(m_fp))
    {
        ::MessageBox(NULL, "End Of Stream!", "DSNetwork", MB_OK);
        return S_FALSE;
    }

    pSample->SetActualDataLength(bytes);
    pSample->SetSyncPoint(FALSE);
    LONGLONG start = mSampleCount++;
    LONGLONG stop  = mSampleCount;
    pSample->SetMediaTime(&start, &stop);

    // 把将要发送出去的Sample数据写到一个本地文件中去
#ifdef _DUMP_SENT
    PBYTE pData8;
    pSample->GetPointer(&pData8);
    if (m_fpDebug)
    {
        fwrite(pData8, 1, pSample->GetActualDataLength(), m_fpDebug);
    }
#endif
    return S_OK;
}
```

程序修改好后，分别编译、注册Filter，然后就可以构建Filter Graph来演示了。运行GraphEdit(SDK安装目录下的Bin\DXUtils\graphedt.exe)，执行菜单命令Graph | Insert Filters，在随后弹出的对话框中打开DirectShow Filters目录，找到MPEG2 TS Source后双击鼠标加入这个Filter，随后会弹出一个标准的文件浏览对话框，读者可以指定一个本地硬盘上的MPEG2 TS文件（本书配套光盘的Media文件夹下提供了一个可以用于演示的MPEG2_TS.mpg文件）。采用同样的方法加入MPEG-2 Multicast SenderFilter和MPEG-2 Multicast Receiver Filter，并且将MPEG2 TS Source与MPEG-2 Multicast Sender相连，如图5.5所示。

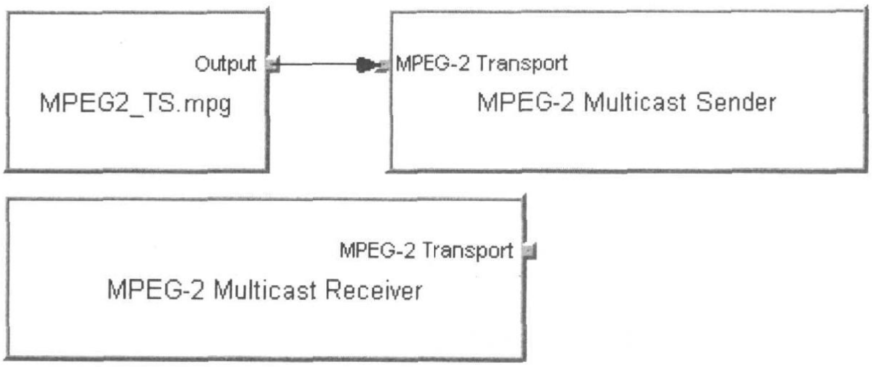


图 5.5　组播发送、接收演示用的 Filter Graph

然后分别打开MPEG-2 Multicast Sender和MPEG-2 Multicast Receiver这两个Filter的属性页，设置IP Address为239.8.8.8，NIC为any，如图5.6所示。设置完后单击Save按钮进行保存，然后就可以关闭属性页窗口了。当然，此时还可以将整个Filter Graph存储为一个GRF文件，方法是执行GraphEdit的菜单命令File | Save As Graph (.GRF)。

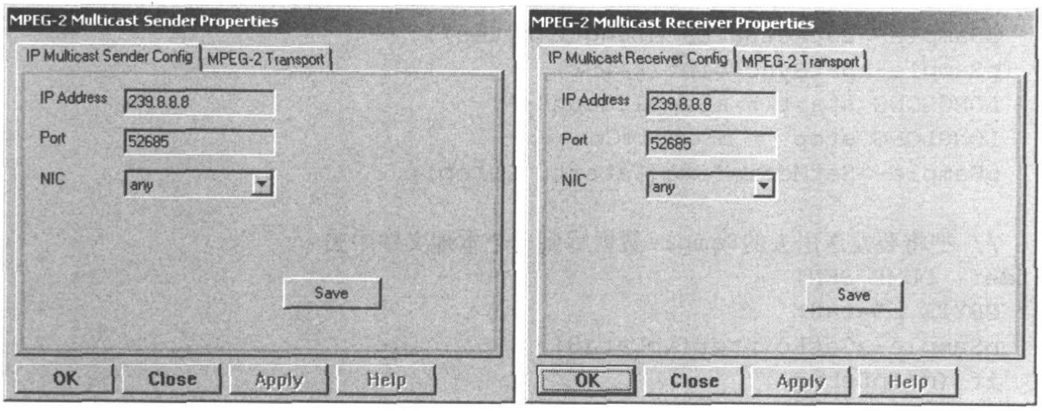


图 5.6　组播发送、接收 Filter 的属性页设置

现在可以运行Filter Graph了——执行GraphEdit的菜单命令Graph | Play。运行一段时间后会弹出一个消息框提示数据流已经发送完毕，此时可以停止Filter Graph的运行（执行菜单命令Graph | Stop）。然后关闭GraphEdit程序，就可以到C:\下找到新生成的MediaSent.mpg和Multicast_Recv_1.mpg两个文件。MediaSent.mpg记录的是MPEG2 TS Source Filter发送的所有Sample数据，Multicast_Recv_1.mpg记录的是MPEG-2 Multicast Receiver Filter从指定的网络组播组中接收到的所有数据。

提示：组播实际上使用UDP协议来传输数据。遗憾的是，UDP传输并不能保证数据包在发送端和接收端次序的一致性，而且数据包在传输过程中还可能丢失，然而DSNetwork例子并没有对此给出解决方案。数据包的乱序到达以及可能的中途丢失，会导致接收端拼装成非正常的MPEG流，最终导致MPEG流解码的异常。基于这方面的原因就不难理解：上述应用演示中，接收端存储下来的MPEG文件Multicast_Recv_1.mpg，它的文件尺寸比发送端的源文件

MPEG2_TS.mpg要小；播放Multicast_Recv_1.mpg时可能还会有马赛克现象。

5.2 VMR-9典型应用

VMR（Video Mixing Renderer）是DirectShow发布的新一代的Video Renderer。目前，VMR有两个版本：VMR-7和VMR-9。VMR-7使用了DirectDraw 7的技术，仅在Windows XP下可用，是该操作系统下默认的Video Renderer；VMR-9使用了最新的Direct3D 9的技术，可在任何安装了DirectX 9运行时库的操作系统上使用，但它在任何操作系统上都不是默认的Video Renderer。

VMR相对于传统的Video Renderer的优势在于：它最大限度地利用了显卡本身的硬件性能；它集成了DirectX中最新的2D和3D技术，使得视频显示的质量和效率都有明显的提升；它有多个输入Pin，可以实现多个视频流的Alpha混合显示等等。

下面，将依次介绍SDK中的几个典型的VMR-9应用例子，包括位图叠加、字符叠加、流混合等。

5.2.1 位图叠加例子BitmapMix

这个例子位于SDK安装目录的Samples\C++\DirectShow\VMR9\BitmapMix下。BitmapMix的程序界面如图5.7所示。

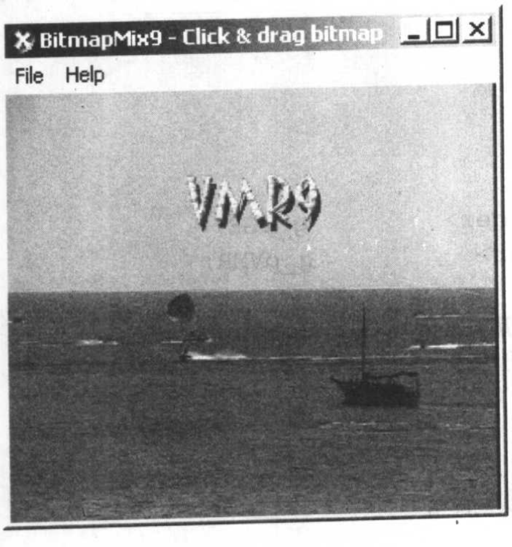


图 5.7 BitmapMix 的程序界面

BitmapMix是一个标准的Win32程序，这个例子实现的功能有：播放用户指定的媒体文件；在播放的同时（抠像）叠加位图，并且实现多个位图交替出现（呈现动画的效果）；支持鼠标拖拉位图，使位图可以叠加在视频窗口的任意位置等。通过这个例子，我们将学习到：

- 如何构建指定使用VMR-9的Filter Graph。

- 如何使用VMR-9的无窗口模式。
- 如何配置3D的应用环境。
- 如何使用3D表面来支持位图的Alpha混合。
- 位图叠加动画效果的实现。
- 鼠标拖动位图到视频窗口任意位置的实现。

1. 如何构建指定使用 VMR-9 的 Filter Graph

VMR-9不是默认的Video Renderer。如果使用常规的智能连接方法，VMR-9将不会被自动创建和连接上。因此，BitmapMix使用了Filter Graph Manager上的IFilterGraph2接口。这个接口并不常用，它是IFilterGraph和IGraphBuilder的扩展接口，主要增加了如下3个接口函数（见表5.1）。

表5.1 IFilterGraph2增加的接口函数

接口函数名	功能描述
AddSourceFilterForMoniker	根据标识对象（如代表音视频采集设备的标识对象）来创建一个Source Filter，并且加入到Filter Graph中
ReconnectEx	将连接着的两个Pin断开，然后指定另外一个媒体类型进行重连（这个接口函数主要是给Filter调用的，应用程序不要使用！）
RenderEx	增强的智能连接函数，可以指定一个Renderer Filter进行连接

BitmapMix例子正是使用上述IFilterGraph2::RenderEx来指定使用VMR-9构建Filter Graph的。

```
//
// BmpMix9.cpp
//

CComPtr<IGraphBuilder>        g_pGB;
CComPtr<IBaseFilter>          g_pVMR;

// 为指定的文件构建播放用的Filter Graph
// 参数hwnd：主窗口句柄
HRESULT StartGraph(HWND hwnd)
{
    HRESULT hr;
    // 其他初始化代码（省略）
    // ……

    // 指定使用VMR-9
    FAIL_RET( RenderFileToVMR9(g_pGB, path, g_pVMR));

    // ……
    return hr;
}

//
```

```
// VMRUtil.h
//
// 指定使用VMR-9构建文件播放用的Filter Graph
// 参数pGB：构建Filter Graph使用的IGraphBuilder接口
// 参数wFileName：用户想要播放的文件
// 参数pRenderer：指定使用的Renderer Filter (即VMR-9)
// 参数bRenderAudio：指示是否需要为播放音频构建Filter链路
HRESULT RenderFileToVMR9(IGraphBuilder *pGB, WCHAR *wFileName,
    IBaseFilter *pRenderer, BOOL bRenderAudio=TRUE)
{
   HRESULT hr=S_OK;
   CComPtr <IPin> pOutputPin;
   CComPtr <IBaseFilter> pSource;
   CComPtr <IBaseFilter> pAudioRenderer;
   CComPtr <IFilterGraph2> pFG;
   // 判断是否是Windows Media文件，如果不是则为它构建Filter Graph
   if (!IsWindowsMediaFile(wFileName))
   {
      // 为指定的文件创建一个Source Filter
      if (FAILED(hr = pGB->AddSourceFilter(wFileName, L"SOURCE",
           &pSource)))
      {
         USES_CONVERSION;
         TCHAR szMsg[MAX_PATH + 128];
         wsprintf(szMsg, TEXT("Failed to add the source filter to the graph!
               hr=0x%x\r\n\r\n")
               TEXT("Filename: %s\0"), hr, W2T(wFileName));
         MessageBox(NULL, szMsg, TEXT("Failed to render file to VMR9"),
            MB_OK | MB_ICONERROR);
         return hr;
      }
      // 得到Source Filter尚未连接的输出Pin (准备往下连接)
      JIF(GetUnconnectedPin(pSource, PINDIR_OUTPUT, &pOutputPin));
   }
   else
{
      // 如果要播放的文件是Windows Media文件，则不构建Filter Graph，并弹出消息
      // 框提示，因为播放这种文件可能需要证书，提供证书认证过程比较繁琐
      MessageBox(NULL, TEXT("Windows Media files (ASF,WMA,WMV) are not
         supported by this application.\r\n\r\n")
         TEXT("For a full example of Windows Media support using the\r\n")
         TEXT("DirectShow WM ASF Reader filter and implementing a key
            provider\r\n")
         TEXT("for Windows Media content, refer to the following SDK
            samples:\r\n\r\n")
         TEXT("\t- ASFCopy\t- AudioBox\r\n\t- Jukebox  \t-
            PlayWndASF\r\n\r\n")
         TEXT("Each of the listed samples provides the necessary extra
            code\r\n")
         TEXT("and links with the required Windows Media libraries.\0"),
         TEXT("Windows Media files are not supported"), MB_OK);
    return E_FAIL;
```

```
    }
    // bRenderAudio标记有效，说明用户想要同时播放音频部分
    if (bRenderAudio)
    {
        // 要使用IFilterGraph2::RenderEx来构建Filter Graph，首先需要创建一个
        // Audio Renderer加入到Filter Graph中
        if (SUCCEEDED(CoCreateInstance(CLSID_DSoundRender, NULL,
            CLSCTX_INPROC_SERVER, IID_IBaseFilter, (void
            **)&pAudioRenderer)))
        {
            JIF(pGB->AddFilter(pAudioRenderer, L"Audio Renderer"));
        }
    }
    // 获得IFilterGraph2接口
    JIF(pGB->QueryInterface(IID_IFilterGraph2, (void **)&pFG));
    // 指定使用Filter Graph中已有的Renderer Filter来构建完整的Filter Graph
    // 注意RenderEx函数的第二个参数是AM_RENDEREX_RENDERTOEXISTINGRENDERERS
    // 如果给定的文件中包含音频内容，则同时构建音频播放的Filter链路
    JIF(pFG->RenderEx(pOutputPin, AM_RENDEREX_RENDERTOEXISTINGRENDERERS,
        NULL));
    // 如果Audio Renderer没有连接上，说明给定的文件中不包含音频内容，这时应该将Audio
    // Renderer从Filter Graph中删除，以防止它干扰文件的播放
    if (pAudioRenderer != NULL)
    {
        IPin *pUnconnectedPin=0;
        // 判断Audio Renderer的输入Pin是否已经连接上了
        HRESULT hrPin = GetUnconnectedPin(pAudioRenderer, PINDIR_INPUT,
            &pUnconnectedPin);
        if (SUCCEEDED(hrPin) && (pUnconnectedPin != NULL))
        {
pUnconnectedPin->Release();
// 如果Audio Renderer的输入Pin没有连接，则将Audio Renderer删除
            hrPin = pGB->RemoveFilter(pAudioRenderer);
        }
    }
    return hr;
}
```

提示：智能连接可以帮助我们很快地构建Filter Graph。但为了指定使用VMR-9，智能连接不是惟一的方法。更一般的做法是，首先创建一个VMR-9 Filter加入到Filter Graph中，然后得到欲连接的输出、输入Pin，再调用IFilterGraph::ConnectDirect进行“手动”连接。

2. 如何使用VMR-9的无窗口模式

传统的Video Renderer在进行视频显示时，内部会创建一个视频窗口。Video Renderer总在自己创建的视频窗口中显示视频图像。开发者如果想让视频显示在应用程序的某个窗口中时，必须改变Video Renderer创建的视频窗口的风格，将其设置成应用程序的视频窗口的子窗口。这个过程似乎比较繁琐！于是，VMR-9提供了一种无窗口模式，即VMR-9内部并不创建视频窗口，而直接将视频图像显示到应用程序指定的视频窗口区域内。

让VMR-9工作在无窗口模式需要进行一些特殊的配置，这个过程中用到的接口包括IVMRFilterConfig9、IVMRWindowlessControl9等。

```
//
// BmpMix9.cpp
//

CComPtr<IVMRFilterConfig9>          g_pFilterConfig;
CComPtr<IVMRWindowlessControl9>     g_pWC;

// 为指定的文件构建播放用的Filter Graph
HRESULT StartGraph(HWND hwnd)
{
    HRESULT hr;
    // 其他初始化代码（省略）
    // ……

    // 创建一个Filter Graph
    FAIL_RET( g_pGB.CoCreateInstance(CLSID_FilterGraph) )
    // 创建一个VMR-9并将其加入到Filter Graph中
    FAIL_RET( g_pVMR.CoCreateInstance(CLSID_VideoMixingRenderer9,
        NULL, CLSCTX_INPROC_SERVER) )
    FAIL_RET( g_pGB->AddFilter(g_pVMR, L"Video Mixing Renderer 9") )
    // 从VMR-9上获取IVMRFilterConfig9接口
    FAIL_RET( g_pVMR->QueryInterface(IID_IVMRFilterConfig9,
        reinterpret_cast<void**>(&g_pFilterConfig)) )
    // 设置VMR-9工作在无窗口模式
    FAIL_RET( g_pFilterConfig->SetRenderingMode( VMR9Mode_Windowless ) )
    // 从VMR-9上获取IVMRWindowlessControl9接口
    FAIL_RET( g_pVMR->QueryInterface(IID_IVMRWindowlessControl9,
        reinterpret_cast<void**>(&g_pWC)) )
    // 将应用程序的主窗口句柄设置给VMR-9，视频图像将直接显示在主窗口的客户区
    FAIL_RET( g_pWC->SetVideoClippingWindow( g_hWnd ) )
    // 设置视频图像显示的窗口区域
    RECT clientRect;
    if( ::GetClientRect(g_hWnd, &clientRect ) )
    {
        FAIL_RET( g_pWC->SetVideoPosition( NULL, &clientRect ) )
    }

    // ……
    return hr;
}
```

VMR-9工作在无窗口模式时，还要注意一下视频窗口的刷新问题。因为Filter Graph在暂停或停止状态时，VMR-9不会自动更新视频窗口中的图像内容。这时，应该在视频窗口的刷新消息响应函数中显式地调用IVMRWindowlessControl9::RepaintVideo进行视频窗口区域的重画。

```
//
// BmpMix9.cpp
```

```
//
// 窗口刷新消息WM_PAINT的响应函数
void OnPaint(HWND hwnd)
{
    HRESULT hr;
    PAINTSTRUCT ps;
    HDC         hdc;
    RECT        rcClient;

    GetClientRect(hwnd, &rcClient);
    hdc = BeginPaint(hwnd, &ps);
    if(g_pWC)
    {
        // 如果使用VMR-9的无窗口模式，当Filter Graph处于暂停或停止状态时，需要显式
        // 地告诉VMR-9进行图像刷新
        if (g_pWC)
            hr = g_pWC->RepaintVideo(hwnd, hdc);
    }
    else
    {
        // 如果没有视频图像，则使用默认的窗口颜色重画整个客户区
        FillRect(hdc, &rcClient, (HBRUSH)(COLOR_WINDOW + 1));
    }
    EndPaint(hwnd, &ps);
}
```

另外，如果当前显示器的显示参数改变，或者视频窗口尺寸改变，也需要通过IVMRWindowlessControl9的接口方法告知VMR-9。

```
//
// BmpMix9.cpp
//
// 主窗口的消息处理过程
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
    lParam)
{
    HRESULT hr;
    switch (message)
    {
        // 其他消息响应（省略）
        // ......
        case WM_PAINT:
            OnPaint(hWnd);
            break;
        case WM_DISPLAYCHANGE:    // 显示器的显示参数改变了
            if( g_pWC )
                g_pWC->DisplayModeChanged();
            break;
        case WM_SIZE:             // 视频窗口的尺寸改变了
            if( g_pWC )
            {
                RECT clientRect;
```

```
                if( ::GetClientRect( g_hWnd, &clientRect ) )
                {
                    g_pWC->SetVideoPosition(NULL, &clientRect );
                }
            }
            break;
    }
    return DefWindowProc(hWnd, message, wParam, lParam);
}
```

3. 如何配置3D的应用环境

VMR-9集成了Direct3D的技术，因此在VMR-9中可以使用3D的一些特性。使用3D特性之前要进行初始化，参见函数D3DStart的实现；在结束使用3D特性之后需要进行反初始化，参见函数ClearD3D的实现。

```
//
// BmpMix9.cpp
//

CComPtr<IDirect3D9>              g_pD3D;
CComPtr<IDirect3DDevice9>        g_pD3DDevice;
const int   NUMBITMAPS           = 3;
CComPtr<IDirect3DSurface9>       g_pSurfaces[NUMBITMAPS];

// 3D应用环境的初始化
HRESULT D3DStart(HWND hWnd)
{
    HRESULT hr;
    ClearD3D();
    // 初始化3D应用环境
    g_pD3D.Release();
    g_pD3D.Attach(Direct3DCreate9( D3D_SDK_VERSION ) );
    if( g_pD3D == NULL )
        return E_FAIL;
    // 得到显卡的显示模式
    D3DDISPLAYMODE d3ddm;
    FAIL_RET( g_pD3D->GetAdapterDisplayMode( D3DADAPTER_DEFAULT,
         &d3ddm ) );
    D3DPRESENT_PARAMETERS d3dpp;
    ZeroMemory( &d3dpp, sizeof(d3dpp) );
    d3dpp.Windowed          = TRUE;
    d3dpp.SwapEffect        = D3DSWAPEFFECT_DISCARD;
    d3dpp.BackBufferFormat = d3ddm.Format;
    d3dpp.hDeviceWindow     = g_hWnd;
    // 创建一个3D设备对象
    FAIL_RET( g_pD3D->CreateDevice(
        D3DADAPTER_DEFAULT,
        D3DDEVTYPE_HAL,
        NULL,
        D3DCREATE_SOFTWARE_VERTEXPROCESSING,
        &d3dpp,
```

```
            &g_pD3DDevice) )
        // 得到欲叠加位图的尺寸
        long lWidth, lHeight;
        if( ! GetImageDimensions( lHeight, lWidth ) )
            return E_FAIL;

        // 为每一个位图创建一个3D离屏表面
        for( int i = 0; i < NUMBITMAPS; ++i )
        {
            FAIL_RET( g_pD3DDevice->CreateOffscreenPlainSurface(
                lWidth, lHeight,
                D3DFMT_X8R8G8B8,
                D3DPOOL_SYSTEMMEM,
                &g_pSurfaces[i], NULL ) );
        }
        // 从资源中将位图装入到各个3D表面
        FAIL_RET( D3DXLoadSurfaceFromResource (
                g_pSurfaces[0],
                NULL,
                NULL,
                GetModuleHandle(NULL),
                MAKEINTRESOURCE(IDB_BITMAP1),
                NULL,
                D3DX_DEFAULT,
                0,
                NULL
                ));
        FAIL_RET( D3DXLoadSurfaceFromResource (
                g_pSurfaces[1],
                NULL,
                NULL,
                GetModuleHandle(NULL),
                MAKEINTRESOURCE(IDB_BITMAP2),
                NULL,
                D3DX_DEFAULT,
                0,
                NULL
                ));
        FAIL_RET( D3DXLoadSurfaceFromResource (
                g_pSurfaces[2],
                NULL,
                NULL,
                GetModuleHandle(NULL),
                MAKEINTRESOURCE(IDB_BITMAP3),
                NULL, // entire image
                D3DX_DEFAULT,
                0,
                NULL
                ));
        return hr;
}
// 3D应用环境的反初始化
```

```
void ClearD3D(void)
{
    for( int i = 0; i < NUMBITMAPS; ++i )
        g_pSurfaces[i] = NULL;
    g_pD3DDevice = NULL;
    g_pD3D = NULL;
}
```

4. 如何使用 3D 表面来支持位图的 Alpha 混合

VMR-9使用IVMRMixerBitmap9接口完成位图与主视频流的Alpha混合。叠加的位图使用一个数据结构VMR9AlphaBitmap来描述，这个数据结构的定义如下：

```
typedef struct _VMR9AlphaBitmap
{
    DWORD dwFlags;   // 控制标记，如指明位图的来源、是否使用色键抠像等
    HDC hdc;         // 指定位图的DC句柄，如果位图由3D表面提供则将hdc指定为NULL
    IDirect3DSurface9 *pDDS;     // 装有位图的3D表面
    RECT rSrc;                   // 指定源区域
    VMR9NormalizedRect rDest;    // 指定位图叠加的区域（使用复合空间坐标）
    FLOAT fAlpha;                // 叠加的Alpha值
    COLORREF clrSrcKey;          // 色键（抠掉的颜色）
    DWORD dwFilterMode;          // 指定一种合成模式
} VMR9AlphaBitmap;
```

我们看到，VMR9AlphaBitmap指定了位图的来源（来自DC或3D表面），指定了位图叠加的区域，还有叠加时用的Alpha值（定义叠加位图的透明度），是否使用色键抠像等。应用程序只需将用VMR9AlphaBitmap描述的位图通过IVMRMixerBitmap9::SetAlphaBitmap设置给VMR-9，就能实现位图在视频图像上的叠加了。

```
//
// BmpMix9.cpp
//

CComPtr<IVMRMixerBitmap9>      g_pMixerBitmap;

// 设置初始叠加的位图
HRESULT SetInitialAlphaBitmap()
{
    HRESULT hr = E_FAIL;
    VMR9AlphaBitmap alphaBitmap;
    // 填写VMR9AlphaBitmap数据结构（叠加第一幅位图）
    FAIL_RET( SetUpAlphaBitmap( alphaBitmap, 0 ) );
    // 将欲叠加的位图设置给VMR-9
    FAIL_RET( g_pMixerBitmap->SetAlphaBitmap( &alphaBitmap ) );
    return hr;
}
// 为指定索引的位图创建VMR9AlphaBitmap描述
HRESULT SetUpAlphaBitmap( VMR9AlphaBitmap& alphaBitmap, int nIndex )
{
    if( g_pMixerBitmap == NULL || g_pSurfaces[nIndex] == NULL )
    {
```

```
        return E_FAIL;
    }
    ZeroMemory( &alphaBitmap, sizeof VMR9AlphaBitmap);
    alphaBitmap.dwFlags   = VMR9AlphaBitmap_EntireDDS |
        VMR9AlphaBitmap_SrcColorKey;
    alphaBitmap.hdc          = NULL; // 使用3D表面提供位图，所以这里赋值为NULL
    alphaBitmap.pDDS         = g_pSurfaces[nIndex]; // 指向一个3D表面
    alphaBitmap.clrSrcKey    = RGB(255,255,255);    // 位图的白色背景抠掉
    // 使用VMR的复合空间（composition space，值域为0.0 - 1.0）坐标
    // g_fCurrentX、g_fCurrentY为当前鼠标的坐标
    // 将位图定位在以鼠标指针为中心的位置
    alphaBitmap.rDest.top    = g_fCurrentY - BMP_SIZE_Y / 2;
    alphaBitmap.rDest.left   = g_fCurrentX - BMP_SIZE_X / 2;
    alphaBitmap.rDest.bottom= g_fCurrentY + BMP_SIZE_Y / 2;
    alphaBitmap.rDest.right  = g_fCurrentX + BMP_SIZE_X / 2;
    alphaBitmap.fAlpha       = ALPHA_VALUE;  // 叠加时使用的Alpha值，默认为0.6
    return S_OK;
}
```

5. 位图叠加动画效果的实现

BitmapMix例子使用了3幅略有差异的位图交替叠加来实现动画效果。每个位图都使用一个3D表面来装载，并且间隔500ms依次设置给VMR-9。这个过程实现如下：

```
//
// BmpMix9.cpp
//
// 主窗口的消息处理过程
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
    lParam)
{
    HRESULT hr;
    switch (message)
    {
     // 其他消息响应（省略）
     // ……
    case WM_TIMER:  // 使用了一个间隔为500ms的定时器
           if( (UINT_PTR) wParam == g_nTimerID )
           {
               // 指向下一个位图
               g_nCurrentImage++;
               g_nCurrentImage %= NUMBITMAPS;
               // 让VMR-9使用新的位图进行叠加
               hr = UpdateAlphaBitmap(g_nCurrentImage);
               if( FAILED( hr ) )
                   return 0;
           }
           break;

    }
   return DefWindowProc(hWnd, message, wParam, lParam);
}
```

```
// 让VMR-9使用新的位图进行叠加
HRESULT UpdateAlphaBitmap(int nIndex)
{
    HRESULT hr = E_FAIL;
    VMR9AlphaBitmap alphaBitmap;
    // 填写VMR9AlphaBitmap数据结构
    FAIL_RET( SetUpAlphaBitmap( alphaBitmap, nIndex ) );
    // 将欲叠加的位图设置给VMR-9
    FAIL_RET( g_pMixerBitmap->SetAlphaBitmap( &alphaBitmap ) );
    return hr;
}
```

6. 鼠标拖动位图到视频窗口任意位置的实现

VMR9AlphaBitmap数据结构的rDest成员可以用来指定位图叠加的区域。如果应用程序能够跟踪鼠标的轨迹，然后相应地更新rDest成员的值，也就能实现叠加位图的随机拖动了。BitmapMix例子的实现如下：

```
//
// BmpMix9.cpp
//
// 主窗口的消息处理过程
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
    lParam)
{
    HRESULT hr;
    switch (message)
    {
        // 其他消息响应（省略）
        // ……
        case WM_MOUSEMOVE:  // 鼠标正在移动
            // 如果此时鼠标左键处于按下状态，则认为是一个拖动操作
            if (wParam & MK_LBUTTON)
                HandleMouseClick(lParam);
            break;
        case WM_LBUTTONDOWN: // 鼠标左键按下
            HandleMouseClick(lParam);
            break;

    }
    return DefWindowProc(hWnd, message, wParam, lParam);
}
// 处理鼠标的单击、拖动事件
void HandleMouseClick(LPARAM lParam)
{
    // 仅在Filter Graph正在运行时支持叠加位图的拖动
    if (!g_bRunning)
        return;
    // 得到鼠标的窗口坐标
    int xPos = GET_X_LPARAM(lParam);
    int yPos = GET_Y_LPARAM(lParam);
    // 得到窗口的客户区矩形
```

```
    RECT rcClient;
    GetClientRect(g_hWnd, &rcClient);
    // 将鼠标的窗口坐标转化成VMR中使用的复合空间坐标
    float fX, fY;
    fX = (float) ((float)xPos / (float)rcClient.right );
    fY = (float) ((float)yPos / (float)rcClient.bottom );
    // 边界检查
    if (fX > 1.0) fX = 1.0;    if (fX < 0.0) fX = 0.0;
    if (fY > 1.0) fY = 1.0;    if (fY < 0.0) fY = 0.0;
    // 由于鼠标移动操作可能会很频繁，为了防止频繁更新叠加位图的位置，仅当鼠标移动了一定
    // 的距离后才认为是一次有效的拖动
    if ((fabs(fX - g_fCurrentX) >= MOVE_TOLERANCE) ||
       (fabs(fY - g_fCurrentY) >= MOVE_TOLERANCE))
    {
       // 将叠加的位图移动到鼠标拖动的新位置
       MoveAlphaBitmap(fX, fY);
    }
}
// 将叠加的位图移动到鼠标拖动的新位置
HRESULT MoveAlphaBitmap(float fX, float fY)
{
    HRESULT hr=S_OK;
    VMR9AlphaBitmap alphaBitmap;
    if( g_pMixerBitmap == NULL || g_pSurfaces[g_nCurrentImage] == NULL )
    {
       return E_FAIL;
    }
    ZeroMemory( &alphaBitmap, sizeof VMR9AlphaBitmap);
    alphaBitmap.dwFlags = VMR9AlphaBitmap_EntireDDS |
        VMR9AlphaBitmap_SrcColorKey;
    alphaBitmap.hdc          = NULL;
    alphaBitmap.pDDS         = g_pSurfaces[g_nCurrentImage];
    alphaBitmap.clrSrcKey    = RGB(255,255,255); // 抠掉白色背景!
    alphaBitmap.rDest.top    = fY - BMP_SIZE_Y / 2;
    alphaBitmap.rDest.left   = fX - BMP_SIZE_X / 2;
    alphaBitmap.rDest.bottom= fY + BMP_SIZE_Y / 2;
    alphaBitmap.rDest.right  = fX + BMP_SIZE_X / 2;
    alphaBitmap.fAlpha       = ALPHA_VALUE;
    // 将新的位图描述设置给VMR-9
    FAIL_RET( g_pMixerBitmap->SetAlphaBitmap( &alphaBitmap ) );
    // 记录当前鼠标拖动的位置
    g_fCurrentX = fX;
    g_fCurrentY = fY;
    return S_OK;
}
```

5.2.2 字符叠加例子Ticker

这个例子位于SDK安装目录的Samples\C++\DirectShow\VMR9\Ticker下。Ticker的程序界面如图5.8所示。

图 5.8 Ticker 的程序界面

Ticker是一个标准的Win32程序，这个例子实现的功能有：播放用户指定的媒体文件；在播放的同时叠加位图或者字符串，并且实现字符从右向左的滚动效果等。通过这个例子，我们将学习到：

- 叠加字符串的两种实现。
- 如何中途取消字符串的叠加。
- 如何实现叠加字符的滚动效果。
- 媒体文件的循环播放。

1. 叠加字符串的两种实现

与BitmapMix例子实现的位图叠加一样，使用VMR-9叠加字符串仍然要使用IVMRMixerBitmap9接口。Ticker例子为叠加字符串提供了两种实现方法。第一种就是首先将字符内容做成一个位图文件（Ticker例子中将字符串"Introducing the Video Mixing Renderer 9, only available with Microsoft DirectX 9!"做成了一个位图文件ticker.bmp），然后像BitmapMix例子一样叠加位图。

```
//
// bitmap.cpp
//
// 将ticker.bmp文件设置给VMR-9进行叠加
// 参数hwndApp是应用程序主窗口
// pWC是IVMRWindowlessControl9接口对象
// pBMP是IVMRMixerBitmap9接口对象
HRESULT BlendApplicationImage(HWND hwndApp)
{
    LONG cx, cy;
    HRESULT hr;
    // 得到视频图像的尺寸
    hr = pWC->GetNativeVideoSize(&cx, &cy, NULL, NULL);
    if (FAILED(hr))
    {
```

```
        Msg(TEXT("GetNativeVideoSize FAILED!  hr=0x%x\r\n"), hr);
        return hr;
    }
    // 从资源中装载位图文件
    HBITMAP hbm = LoadBitmap(ghInst, MAKEINTRESOURCE(IDR_TICKER));
    // 创建一个与主窗口DC兼容的内存DC
    HDC hdc = GetDC(hwndApp);
    HDC hdcBmp = CreateCompatibleDC(hdc);
    ReleaseDC(hwndApp, hdc);
    // 把欲叠加的位图选入内存兼容DC
    BITMAP bm;
    HBITMAP hbmOld;
    GetObject(hbm, sizeof(bm), &bm);
    hbmOld = (HBITMAP)SelectObject(hdcBmp, hbm);
    // 为将要叠加的位图填写一个VMR9AlphaBitmap描述信息
    VMR9AlphaBitmap bmpInfo;
    ZeroMemory(&bmpInfo, sizeof(bmpInfo) );
    bmpInfo.dwFlags = VMRBITMAP_HDC;     // 通过DC来提供叠加的位图
    bmpInfo.hdc = hdcBmp;                // 装有欲叠加位图的DC句柄
    // 保存位图的实际宽度
    g_nImageWidth = bm.bmWidth;
    // 将位图宽度转换成复合空间坐标
    g_fBitmapCompWidth = (float)g_nImageWidth / (float)cx;
    // 将位图初始叠加在右下角
    SetRect(&bmpInfo.rSrc, 0, 0, g_nImageWidth, bm.bmHeight);
    bmpInfo.rDest.left   = 1.0f;
    bmpInfo.rDest.right  = 1.0f + g_fBitmapCompWidth;
    bmpInfo.rDest.top    = (float)(cy - bm.bmHeight) / (float)cy -
        EDGE_BUFFER;
    bmpInfo.rDest.bottom = 1.0f - EDGE_BUFFER;
    // 保存位图叠加的区域（以后可能改变以实现滚动效果）
    g_rDest = bmpInfo.rDest;
    // 叠加的Alpha值，默认为0.6
    bmpInfo.fAlpha = TRANSPARENCY_VALUE;
    // 设置色键抠像（把字符位图的白色背景抠掉！）
    SetColorRef(bmpInfo);
    // 把欲叠加的位图设置给VMR-9
    hr = pBMP->SetAlphaBitmap(&bmpInfo);
    if (FAILED(hr))
    {
        Msg(TEXT("SetAlphaBitmap FAILED!  hr=0x%x\r\n\r\n%s\0"),
            hr, STR_VMR_DISPLAY_WARNING);
    }
    // 把欲叠加的位图从内存DC中选出，并释放GDI资源
    DeleteObject(SelectObject(hdcBmp, hbmOld));
    DeleteObject(hbm);
    DeleteDC(hdcBmp);
    return hr;
}
// 设置色键抠像功能
void SetColorRef(VMR9AlphaBitmap& bmpInfo)
{
```

```
    bmpInfo.clrSrcKey = RGB(255, 255, 255);      // 将白色抠掉
    bmpInfo.dwFlags |= VMRBITMAP_SRCCOLORKEY;    // 修改控制标记
}
```

第二种字符叠加的方法从本质上来说仍然是位图叠加，只是比第一种方法要稍微灵活一点——让用户设定欲叠加的字符内容，然后应用程序在内存中为这个字符串创建一个位图，再设置给VMR-9进行叠加。使用这种方法，叠加字符的内容随时可以改变，而且还可以定制字符串的字体类型、颜色、大小等属性。

```
//
// bitmap.cpp
//
// 将用户想要叠加的字符内容设置给VMR-9进行叠加
// 参数hwndApp是应用程序主窗口
// 参数szNewText是将要叠加的字符串
HRESULT BlendApplicationText(HWND hwndApp, TCHAR *szNewText)
{
    LONG cx, cy;
    HRESULT hr;
    // 得到视频图像的尺寸
    hr = pWC->GetNativeVideoSize(&cx, &cy, NULL, NULL);
    if (FAILED(hr))
      return hr;
    // 创建一个与主窗口DC兼容的内存DC
    HDC hdc = GetDC(hwndApp);
    HDC hdcBmp = CreateCompatibleDC(hdc);
    // 选入一个用户设定的字体
    HFONT hOldFont = (HFONT) SelectObject(hdcBmp, g_hFont);
    // 判断字符串的长度，以及为它创建一个位图需要的位图尺寸
    int nLength, nTextBmpWidth, nTextBmpHeight;
    SIZE sz={0};
    nLength = (int) _tcslen(szNewText);
    GetTextExtentPoint32(hdcBmp, szNewText, nLength, &sz);
    nTextBmpHeight = sz.cy;
    nTextBmpWidth  = sz.cx;
    // 在内存中创建一个位图
    HBITMAP hbm = CreateCompatibleBitmap(hdc, nTextBmpWidth,
        nTextBmpHeight);
    ReleaseDC(hwndApp, hdc);
    // 把欲叠加的位图选入内存兼容DC
    BITMAP bm;
    HBITMAP hbmOld;
    GetObject(hbm, sizeof(bm), &bm);
    hbmOld = (HBITMAP)SelectObject(hdcBmp, hbm);
    // 初始化位图
    RECT rcText;
    SetRect(&rcText, 0, 0, nTextBmpWidth, nTextBmpHeight);
    SetBkColor(hdcBmp, RGB(255, 255, 255)); // 位图的背景为白色
    SetTextColor(hdcBmp, g_rgbColors);       // 字符颜色由用户设定
    // 在位图上画出字符内容
    TextOut(hdcBmp, 0, 0, szNewText, nLength);
    // 为将要叠加的位图填写一个VMR9AlphaBitmap描述信息
```

```
    VMR9AlphaBitmap bmpInfo;
    ZeroMemory(&bmpInfo, sizeof(bmpInfo) );
    bmpInfo.dwFlags = VMRBITMAP_HDC;     // 通过DC来提供叠加的位图
    bmpInfo.hdc = hdcBmp;                // 装有欲叠加位图的DC句柄
    // 保存位图的实际宽度
    g_nImageWidth = bm.bmWidth;
    // 将位图宽度转换成复合空间坐标
    g_fBitmapCompWidth = (float)g_nImageWidth / (float)cx;
    // 将位图初始叠加在右下角
    bmpInfo.rDest.left  = 1.0f;
    bmpInfo.rDest.right = 1.0f + g_fBitmapCompWidth;
    bmpInfo.rDest.top = (float)(cy - bm.bmHeight) / (float)cy - EDGE_BUFFER;
    bmpInfo.rDest.bottom = 1.0f - EDGE_BUFFER;
    bmpInfo.rSrc = rcText;
    // 保存位图叠加的区域（以后可能改变以实现滚动效果）
    g_rDest = bmpInfo.rDest;
    // 叠加的Alpha值，默认为0.6
    bmpInfo.fAlpha = TRANSPARENCY_VALUE;
    // 设置色键抠像（把字符位图的白色背景抠掉！）
    SetColorRef(bmpInfo);
    // 把欲叠加的位图设置给VMR-9
    hr = pBMP->SetAlphaBitmap(&bmpInfo);
    if (FAILED(hr))
        Msg(TEXT("SetAlphaBitmap FAILED!  hr=0x%x\r\n\r\n%s\0"), hr,
            STR_VMR_DISPLAY_WARNING);
    // 把欲叠加的位图从内存DC中选出，并释放GDI资源
    DeleteObject(SelectObject(hdcBmp, hbmOld));
    SelectObject(hdc, hOldFont);
    DeleteObject(hbm);
    DeleteDC(hdcBmp);
    return hr;
}
```

2. 如何中途取消字符串的叠加

执行Ticker程序的菜单命令Ticker | Disable/Enable，可以将正在进行的字符叠加功能关闭；再次执行该菜单命令，可以重新启用字符叠加功能。Ticker程序是怎么来实现的呢？

其实，VMR-9中描述欲叠加位图的数据结构VMR9AlphaBitmap有一个成员dwFlags，这是一个控制标记，将它赋值为VMRBITMAP_DISABLE即可关闭叠加功能，将这个标记去除又可恢复叠加功能。

```
//
// bitmap.cpp
//
// 将VMR-9的位图叠加功能打开或关闭
HRESULT DisableTicker(DWORD dwFlags)
{
    HRESULT hr;
    VMR9AlphaBitmap bmpInfo={0};
    // 得到VMR-9中当前使用的叠加位图描述参数
    hr = pBMP->GetAlphaBitmapParameters(&bmpInfo);
```

```
    if (FAILED(hr))
        Msg(TEXT("GetAlphaBitmapParameters FAILED!  hr=0x%x\r\n"), hr);
    // 如果调用者想要关闭叠加功能……
    if (dwFlags & MARK_DISABLE)
    {
        // 通过VMR9AlphaBitmap的控制标记关闭叠加功能
        bmpInfo.dwFlags = VMRBITMAP_DISABLE;
        // 更新主程序各控制菜单项的状态
        EnableMenuItem(ghMenu, ID_SLIDE,   MF_GRAYED);
        EnableMenuItem(ghMenu, ID_TICKER_STATIC_IMAGE, MF_GRAYED);
        EnableMenuItem(ghMenu, ID_TICKER_DYNAMIC_TEXT, MF_GRAYED);
        EnableMenuItem(ghMenu, ID_SET_FONT, MF_GRAYED);
        EnableMenuItem(ghMenu, ID_SET_TEXT, MF_GRAYED);
        // 将修正的VMR9AlphaBitmap结构设置给VMR-9
        hr = pBMP->UpdateAlphaBitmapParameters(&bmpInfo);
        if (FAILED(hr))
            Msg(TEXT("UpdateAlphaBitmapParameters FAILED to disable ticker!
                 hr=0x%x\r\n"), hr);
    }
    else  // 如果调用者想要重新启用叠加功能
    {
        // 根据叠加位图的不同来源调用相应的函数
        if (g_dwTickerFlags & MARK_STATIC_IMAGE)
            hr = BlendApplicationImage(ghApp);
        else
            hr = BlendApplicationText(ghApp, g_szAppText);
        // 更新主程序各控制菜单项的状态
        EnableMenuItem(ghMenu, ID_SLIDE,   MF_ENABLED);
        EnableMenuItem(ghMenu, ID_TICKER_STATIC_IMAGE, MF_ENABLED);
        EnableMenuItem(ghMenu, ID_TICKER_DYNAMIC_TEXT, MF_ENABLED);
        EnableMenuItem(ghMenu, ID_SET_FONT, MF_ENABLED);
        EnableMenuItem(ghMenu, ID_SET_TEXT, MF_ENABLED);
    }
    return hr;
}
```

3. 如何实现叠加字符的滚动效果

Ticker程序可以让叠加的字符从右向左滚动，周而复始。这是怎么实现的呢？很自然的想法是，使用一个定时器，不断地更新叠加位图的起始位置坐标；坐标位置以相同的时间间隔从右向左步步推进，表现出来的字符叠加效果自然就是滚动的了。Ticker程序就是这么做的！

```
//
// Ticker.cpp
//
// 主窗口消息处理函数
LRESULT CALLBACK WndMainProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
    lParam)
{
    switch(message)
    {
```

```
        // 其他消息处理（省略）
        // ......
        case WM_COMMAND:  // 菜单命令响应
            switch(wParam)
            {
                case ID_SLIDE:   // 打开/关闭字符滚动
                    FlipFlag(MARK_SLIDE);
                    SlideTicker(g_dwTickerFlags);
                    break;

            }
            break;

        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
        // ......
    }
    return DefWindowProc(hWnd, message, wParam, lParam);
}

//
// bitmap.cpp
//
// 标记置反
void FlipFlag(DWORD dwFlag)
{
    // 如果指定的标记有效，则将它设置成无效
    if (g_dwTickerFlags & dwFlag)
        g_dwTickerFlags &= ~dwFlag;
    // 如果指定的标记无效，则将它设置成有效
    else
        g_dwTickerFlags |= dwFlag;
}
// 触发叠加的字符滚动或者将这个功能关闭
void SlideTicker(DWORD dwFlags)
{
    if (dwFlags & MARK_SLIDE)
{
    // 启动一个定时器，定时器消息的处理函数为TimerProc
    // 定时间隔SLIDE_TIMEOUT定义为125ms
    gnSlideTimer = (int) SetTimer(NULL, SLIDE_TIMER, SLIDE_TIMEOUT,
         TimerProc);
    CheckMenuItem(ghMenu, ID_SLIDE, MF_CHECKED);
    }
    else
{
    // 关闭定时器以停止字符的滚动叠加
    if (gnSlideTimer)
    {
        KillTimer(NULL, gnSlideTimer);
        gnSlideTimer=0;
    }
    CheckMenuItem(ghMenu, ID_SLIDE, MF_UNCHECKED);
    // 重新设置字符叠加的位置
    ResetBitmapPosition();
```

```
    }
}
// 定时器消息的处理函数
VOID CALLBACK TimerProc(
  HWND hwnd,              // 窗口句柄
  UINT uMsg,              // WM_TIMER消息
  UINT_PTR idEvent,       // 定时器的ID
  DWORD dwTime            // 当前的系统时间
)
{
    HandleSlide();
}
// 处理字符滚动的函数
// const float EDGE_BUFFER = 0.04f;
// const float SLIDE_VALUE = 0.05f;
void HandleSlide(void)
{
    HRESULT hr;
    VMR9AlphaBitmap bmpInfo={0};
    // 得到VMR-9中当前使用的叠加位图描述参数
    hr = pBMP->GetAlphaBitmapParameters(&bmpInfo);
    if (FAILED(hr))
        Msg(TEXT("GetAlphaBitmapParameters FAILED!  hr=0x%x\r\n"), hr);
    // 将字符叠加的坐标位置向右移
    bmpInfo.rDest.left  -= SLIDE_VALUE;
    bmpInfo.rDest.right -= SLIDE_VALUE;
    // 如果整个字符串从屏幕上滚动一次后消失，则重新定位到右下角开始下一次滚动
    if ((bmpInfo.rDest.right <= EDGE_BUFFER))
    {
        bmpInfo.rDest.left = 1.0f;
        bmpInfo.rDest.right = 1.0f + g_fBitmapCompWidth;
    }
    SetColorRef(bmpInfo);
    // 将修正的VMR9AlphaBitmap结构设置给VMR-9
    hr = pBMP->UpdateAlphaBitmapParameters(&bmpInfo);
}
// 将字符滚动效果关闭时重新设置字符叠加的位置
void ResetBitmapPosition(void)
{
    HRESULT hr;
    VMR9AlphaBitmap bmpInfo={0};
    // 得到VMR-9中当前使用的叠加位图描述参数
    hr = pBMP->GetAlphaBitmapParameters(&bmpInfo);
    if (FAILED(hr))
        Msg(TEXT("GetAlphaBitmapParameters FAILED!  hr=0x%x\r\n"), hr);
    // 将字符叠加的位置重新定位到右下角（尚未露出来！）
    bmpInfo.rDest.left  = g_rDest.left;
    bmpInfo.rDest.right = g_rDest.right;
    SetColorRef(bmpInfo);
    // 将修正的VMR9AlphaBitmap结构设置给VMR-9
    hr = pBMP->UpdateAlphaBitmapParameters(&bmpInfo);
}
```

4. 媒体文件的循环播放

一个媒体文件播放完成之后，Filter Graph Manager会发出一个EC_COMPLETE事件。如果应用程序注册了一个用于获取DirectShow事件的窗口消息（当Filter Graph Manager有事件需要应用程序处理时发出该窗口消息进行通知），在该窗口消息的响应函数中就可以对EC_COMPLETE事件作出一些处理——将Filter Graph重新定位到流的开头，然后重新播放。

```
//
// Ticker.cpp
//
// 为用户选定的媒体文件构建播放用的Filter Graph
// #define WM_GRAPHNOTIFY  WM_USER+13
HRESULT PlayMovieInWindow(LPTSTR szFile)
{
    JIF(CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER,
        IID_IGraphBuilder, (void **)&pGB));
    if(SUCCEEDED(hr))
    {
    // 获取IMediaEventEx接口
    JIF(pGB->QueryInterface(IID_IMediaEventEx, (void **)&pME));
     // 注册一个事件通知消息WM_GRAPHNOTIFY
    JIF(pME->SetNotifyWindow((OAHWND)ghApp, WM_GRAPHNOTIFY, 0));
    // 其他代码省略
    // ……
    }
    return hr;
}
// 主窗口消息处理函数
LRESULT CALLBACK WndMainProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
     lParam)
{
    switch(message)
    {
        // 其他消息处理（省略）
        // ……
        case WM_GRAPHNOTIFY:  // Filter Graph Manager有事件需要应用程序处理
            HandleGraphEvent();
            break;

        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
        // ……
    }
    return DefWindowProc(hWnd, message, wParam, lParam);
}
// DirectShow事件处理函数
HRESULT HandleGraphEvent(void)
{
    LONG evCode, evParam1, evParam2;
    HRESULT hr=S_OK;
    if (!pME)
        return S_OK;
    // 使用while循环处理事件队列中缓存的所有事件
    while(SUCCEEDED(pME->GetEvent(&evCode, (LONG_PTR *) &evParam1,
```

```
        (LONG_PTR *) &evParam2, 0)))
    {
        // 释放该事件占用的资源
        hr = pME->FreeEventParams(evCode, evParam1, evParam2);
        // 判断事件类型是否表示媒体文件播放结束了
        if(EC_COMPLETE == evCode)
        {
            LONGLONG pos=0;
            // 将Filter Graph定位到流的开头继续播放
            hr = pMS->SetPositions(&pos, AM_SEEKING_AbsolutePositioning ,
                NULL, AM_SEEKING_NoPositioning);
            if (FAILED(hr))
            {
                // 如果上述定位失败，则将Filter Graph停止以后再重新启动
                hr = pMC->Stop();
                hr = pMC->Run();
            }
        }
    }
    return hr;
}
```

5.2.3 流混合例子Blender

这个例子位于SDK安装目录的Samples\C++\DirectShow\VMR9\Blender下。Blender的程序界面如图5.9所示。

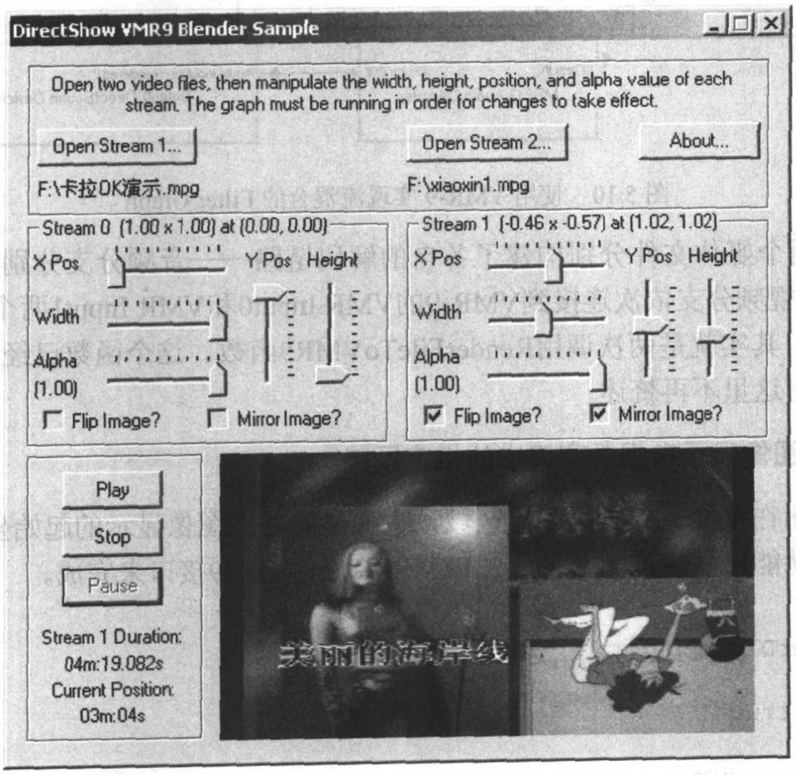


图 5.9　Blender 的程序界面

Blender是一个基于对话框的MFC程序，这个例子实现的功能有：同时播放两个媒体文件；将两个媒体文件的视频内容做Alpha混合，并且可以随时调整视频图像叠加的位置，对视频图像作镜像或上下颠倒处理等。通过这个例子，我们将学习到：

- 流混合的实现方案。
- 如何将图像显示在视频窗口中的指定区域。
- 如何对视频图像作镜像或上下颠倒处理。
- 如何调整叠加视频的Alpha值。
- 如何注册应用程序创建的Filter Graph（让GraphEdit查看）。

1. 流混合的实现方案

VMR-9有多个输入Pin，可以让多个视频流同时进入，然后合成显示。这种Filter Graph的连接如图5.10所示。

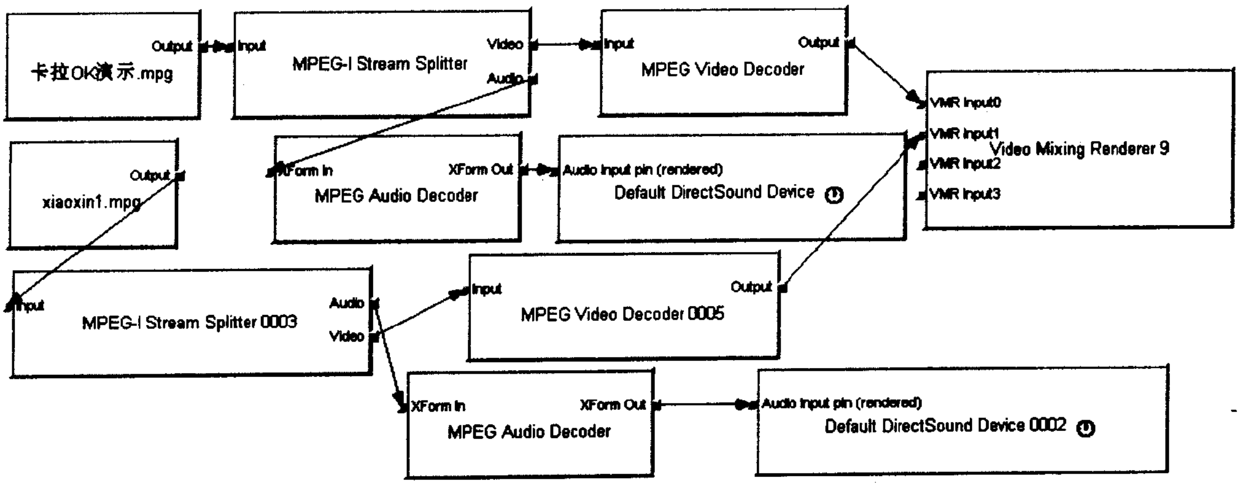


图 5.10　使用 VMR-9 实现流混合的 Filter Graph

图5.10中两个媒体文件分别构建了各自的解码链路——音频分支分别使用一个Audio Renderer播放，视频分支依次连接到VMR-9的VMR Input0和VMR Input1两个输入Pin上。在程序中实现时，其实就是两次调用RenderFileToVMR9函数。这个函数已经在BitmapMix例子中介绍过了，这里不再赘述。

2. 如何将图像显示在视频窗口中的指定区域

将图像显示在视频窗口中的指定区域，也就是要调整图像显示的起始坐标以及图像的宽和高。这个功能可以通过VMR-9上的IVMRMixerControl9接口来完成。

```
//
// BlenderDlg.cpp
//
typedef struct
{
    FLOAT    xPos,  yPos;
    FLOAT    xSize, ySize;
    FLOAT    Alpha;
```

```
    BOOL     bFlipped, bMirrored;
} STRM_PARAM;
// 更新指定输入流的显示区域
// pMix是IVMRMixerControl9接口对象
HRESULT CBlenderDlg::UpdatePinPos(int nStreamID)
{
    HRESULT hr=S_OK;
    // 得到指定输入流新设定的参数值，包括X/Y坐标，图像的宽高等
    STRM_PARAM* p = &strParam[nStreamID];
    // 将图像的显示区域转换成VMR9NormalizedRect结构描述
    VMR9NormalizedRect r = {p->xPos, p->yPos, p->xPos + p->xSize, p->yPos
        + p->ySize};
    // 镜像处理（暂时略过）
    if (strParam[nStreamID].bMirrored)
    {
        // ……
    }
    // 上下颠倒处理（暂时略过）
    if (strParam[nStreamID].bFlipped)
    {
        // ……
    }
    // 在界面上显示更新过的坐标信息
    DisplayCoordinates(nStreamID, r);
    // 将指定输入流调整过的显示区域设置给VMR-9
    if(pMix)
        hr = pMix->SetOutputRect(nStreamID, &r);
    return hr;
}
```

值得注意的是，VMR-9使用的是复合空间坐标系统，坐标值采用浮点数来表示，取值范围为0~1.0。因此，在用户调整显示参数之后，程序中应该进行一些转换（以调整第一个流的X坐标为例）：

```
//
// BlenderDlg.cpp
//
// 处理水平滑动条的消息
void CBlenderDlg::HandleHorizontalTrackbar(WPARAM wParam, LPARAM lParam)
{
    if (wParam != SB_ENDSCROLL)
    {
        // 首先判断是哪个滑动条发出的这个消息
        HWND hwnd = (HWND) lParam;
        int nID = ::GetWindowLong(hwnd, GWL_ID);
        CSliderCtrl *pSlider = (CSliderCtrl *)GetDlgItem(nID);
        // 得到该滑动条当前的位置（滑动条的取值范围为0~100）
        DWORD dwPosition = pSlider->GetPos();
        // 将滑动条当前的位置转换成复合空间坐标值
        FLOAT fPos = (FLOAT) dwPosition * (FLOAT) 0.01;
        switch (nID)
        {
            case IDC_SLIDER_X:
                strParam[0].xPos = fPos;   // 更新X坐标
                UpdatePinPos(0);           // 将新坐标设置给VMR-9
```

```
                break;
            // 其他控件处理（省略）
            // ……
        }
    }
}
```

3. 如何对视频图像作镜像或上下颠倒处理

VMR-9对显示的视频图像作镜像或上下颠倒处理，实现起来非常简单。镜像处理——将图像在水平方向上的起始坐标和结束坐标互换；上下颠倒处理——将图像在垂直方向上的起始坐标和结束坐标互换。仍然来看UpdatePinPos函数的实现：

```
//
// BlenderDlg.cpp
//
// 更新指定输入流的显示区域
HRESULT CBlenderDlg::UpdatePinPos(int nStreamID)
{
    // 其他代码省略
    // ……

    // 镜像处理
    if (strParam[nStreamID].bMirrored)
    {
        // 水平方向上的起始坐标和结束坐标互换
        float fLeft = strParam[nStreamID].xPos;
        float fRight = strParam[nStreamID].xPos + strParam[nStreamID].xSize;
        r.left = fRight;
        r.right = fLeft;
    }
    // 上下颠倒处理
    if (strParam[nStreamID].bFlipped)
    {
        // 垂直方向上的起始坐标和结束坐标互换
        float fTop = strParam[nStreamID].yPos;
        float fBottom = strParam[nStreamID].yPos +
             strParam[nStreamID].ySize;
        r.top = fBottom;
        r.bottom = fTop;
}

// ……
    // 将指定输入流调整过的显示区域设置给VMR-9
    if(pMix)
        hr = pMix->SetOutputRect(nStreamID, &r);
    return hr;
}
```

4. 如何调整叠加视频的 Alpha 值

调整叠加视频的Alpha值同样可以通过VMR-9上的IVMRMixerControl9接口来完成。参见Blender程序中UpdatePinAlpha函数的实现：

```
//
// BlenderDlg.cpp
//
// 调整指定输入流的Alpha值
// pMix是IVMRMixerControl9接口对象
HRESULT CBlenderDlg::UpdatePinAlpha(int nStreamID)
{
    HRESULT hr=S_OK;
    // 得到指定输入流新设定的参数值，包括X/Y坐标，Alpha值等
    STRM_PARAM* p = &strParam[nStreamID];
    // 将指定输入流调整过的Alpha值设置给VMR-9
    if(pMix)
        hr = pMix->SetAlpha(nStreamID, p->Alpha);
    // 在界面上显示新的Alpha值
    if (SUCCEEDED(hr))
        DisplayAlpha(nStreamID);
    return hr;
}
```

5. 如何注册应用程序创建的 Filter Graph（让 GraphEdit 查看）

参见源文件中使用REGISTER_FILTERGRAPH宏包裹的部分。这个过程大致为：在Filter Graph创建起来的时候调用AddGraphToRot函数将其注册；在Filter Graph销毁之前将其反注册（Blender例子遗漏了！）。

```
//
// BlenderDlg.cpp
//
// 创建Filter Graph
HRESULT CBlenderDlg::InitializeVideo(void)
{
    JIF(CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER,
       IID_IGraphBuilder, (void **)&pGB));
    if(SUCCEEDED(hr))
    {
        // 将成功创建的Filter Graph注册
#ifdef REGISTER_FILTERGRAPH
        m_dwGraphRegister = 0;
        hr = AddGraphToRot(pGB, &m_dwGraphRegister);
        if (FAILED(hr))
            Msg(TEXT("Failed to register filter graph with ROT!  hr=0x%x"),
                hr);
#endif
        // 其他代码省略
        // ……
    }
    return hr;
}
// 销毁Filter Graph
HRESULT CBlenderDlg::FreeDirectShow(void)
{
        // 将当前的Filter Graph反注册
#ifdef REGISTER_FILTERGRAPH
        RemoveGraphFromRot(m_dwGraphRegister);
```

```
#endif
        // 释放所有接口，销毁Filter Graph（省略）
        // ……
        return hr;
}
// --- Filter Graph的注册函数和反注册函数的实现 ---
// 编译时必须定义了REGISTER_FILTERGRAPH宏才有意义！
#ifdef REGISTER_FILTERGRAPH
HRESULT CBlenderDlg::AddGraphToRot(IUnknown *pUnkGraph, DWORD
     *pdwRegister)
{
    IMoniker * pMoniker;
    IRunningObjectTable *pROT;
    if (FAILED(GetRunningObjectTable(0, &pROT)))
    {
        return E_FAIL;
    }
    WCHAR wsz[128];
    wsprintfW(wsz, L"FilterGraph %08x pid %08x", (DWORD_PTR)pUnkGraph,
       GetCurrentProcessId());
    HRESULT hr = CreateItemMoniker(L"!", wsz, &pMoniker);
    if (SUCCEEDED(hr))
    {
        // 注意：Register函数使用ROTFLAGS_REGISTRATIONKEEPSALIVE参数，
        // 可以保证GraphEdit能够多次打开同一个注册的Filter Graph
        hr = pROT->Register(ROTFLAGS_REGISTRATIONKEEPSALIVE, pUnkGraph,
             pMoniker, pdwRegister);
        pMoniker->Release();
    }
    pROT->Release();
    return hr;
}
void CBlenderDlg::RemoveGraphFromRot(DWORD pdwRegister)
{
    IRunningObjectTable *pROT;
    if (SUCCEEDED(GetRunningObjectTable(0, &pROT)))
    {
        pROT->Revoke(pdwRegister);
        pROT->Release();
    }
}
#endif
```

运行Blender程序，选择两个媒体文件后开始播放。随后运行GraphEdit（即SDK目录下的\Bin\DXUtils\graphedt.exe），执行菜单命令File | Connect to Remote Graph，在随后弹出的对话框中选择一个注册项，单击OK按钮后即可看到如图5.10所示的Filter Graph了。

提示： *使用GraphEdit连接应用程序注册的Filter Graph时可能会出错。没关系！只要你有足够的耐心，将出错消息框关闭以后再试，直到连接成功为止。*

FAQ

DirectShow常见问题解答

A.1 一般性问题

1. DirectShow支持哪些操作系统？

DirectShow支持Windows 9x/2000/Me/XP等操作系统。

2. DirectShow编程需要多少COM知识？

对于DirectShow应用程序开发者来说，你需要知道COM组件的基本原理，包括如何创建它们，如何得到并使用它们的接口，如何管理引用计数等。对于Filter开发者来说，你需要更多的COM知识，即如何实现一个COM组件。

3. DirectShow支持哪些媒体格式？

DirectShow是一个开放性架构，只要系统中安装了分析器和解码器，它在理论上能支持任何一种格式。目前，通过微软公司提供的一些Filter，DirectShow支持的文件类型：WMA、WMV、ASF、MPEG、AVI、QuickTime（2.0以下版本）、WAV、AIFF、AU、SND、MIDI等；压缩格式：WMV、WMA、MPEG-4 V1、MPEG-4 V3、Sipro Labs ACELP、MP3、DV、MPEG-1、MJPEG、Cinepak等。

4. 有没有DirectShow兼容的硬件设备列表？

没有。DirectShow自动使用硬件设备的DirectDraw和DirectSound操作能力。如果硬件不支持这些特性，那么DirectShow将使用GDI函数进行视频显示，使用waveOut等API函数进行声音的播放。

5. 开发DirectShow应用程序应该使用哪种编程语言？

DirectShow主要是为C++开发者设计的。使用VB，你就只能使用有限的一些DirectShow API。DirectShow支持基于脚本的DVD应用程序（使用微软公司的JScript或VB Script语言）。当然，使用Borland公司的Delphi也能开发DirectShow应用程序，但是SDK文档并没有相关的介绍。

6. DirectShow开发需要什么样的编译器？

只要开发环境配置得当，任何能够生成COM组件的编译器都能胜任DirectShow开发。

7. DirectShow与Windows Media有何关联？

DirectShow提供了两个Filter：WM ASF Reader和WM ASF Writer，来支持ASF文件的播放和生成。

8. DirectShow与DirectX中的其他成员有何关联？

只要硬件支持，DirectShow内部自动使用DirectSound和DirectDraw技术。传统的Video Renderer和Overlay Mixer Filter使用DirectDraw 3和DirectDraw 5，Video Mixing Renderer 7（仅在Windows XP下适用）使用DirectDraw 7，Video Mixing Renderer 9使用最新的Direct 3D 9。开发DirectShow应用程序，没有必要组合使用DirectX中其他成员的API，尽管你可以这么做。

9. DirectShow与微软公司的ActiveMovie有何关系？

ActiveMovie是DirectShow的前身。ActiveMovie这个术语已经不再使用，但一些API名字仍然保留着AM前缀，如AM_MEDIA_TYPE、IAMVideoAccelerator等。

10. DirectShow仅限于多媒体应用领域吗？

DirectShow集成的组件主要是为处理流式的音视频数据而设计的。但是，DirectShow架构也有很多被用于处理其他类型的流式数据的成功案例。

11. GraphEdit开放源代码吗？GraphEdit能再发布吗？

GraphEdit不开放源代码，Graphedt.exe不可再发布。

12. DMO会代替DirectShow Filter吗？

DMO（DirectX Media Object）可以在DirectShow应用程序中使用。对于编码器、解码器、效果处理，建议你使用DMO来代替DirectShow Filter。但在其他一些场合，选择DirectShow Filter会更合适一点。更多的DMO相关信息，SDK文档中有专门的章节介绍。

13. 哪里能获得更多的DirectShow相关信息？

可以访问MSDN的DirectX开发者中心（http://go.microsoft.com/fwlink/?LinkId=10138），或者加入DirectX开发论坛（http://go.microsoft.com/fwlink/?LinkId=10137）。

14. 开发DirectShow应用程序或Filter时碰到了问题，在哪里能获得技术支持？

可以打电话(800) 936-5800或900-555-2300获得帮助，或者访问MSDN，或直接向微软公司提交问题（网址是http://support.microsoft.com/support/webresponse.asp）。

15. 有没有相关书籍介绍DirectShow?

目前，DirectShow的专业著作在美国仅有两本：Michael Linetsky的《Programming Microsoft DirectShow》和Mark D. Pesce的《Programming Microsoft DirectShow for Digital Video and Television》。另外，中国有一本：《DirectShow开发指南》，作者是陆其明。

16. 如果我有些建议可以改进DirectShow，我该怎么办?

请给微软公司去信：askdshow@microsoft.com。

17. 有时候，在用Windows Media Player播放一个AVI文件时，发现只有声音，看不到视频内容，这是为什么?

出现这种情况，很有可能是你播放的文件中的视频部分使用了一种特殊的编码器生成，但是在系统中没有注册相应的解码器。AVI是一种通用的文件格式，它本身并没有限定其中包含的音频和视频的格式。所以，在播放经过特殊编码的AVI文件之前，请确认你已经安装了相应的解码器。（典型的情况：播放采用DivX压缩的AVI文件时，需要预先安装一个DivX Decoder Filter。）

18. 如何获知指定的一个文件的编码格式?

在文件浏览器中右击该文件，在弹出的菜单中选择Properties命令，然后在弹出的对话框中单击Summary标签就可以看到。

A.2 编程问题

19. 如何创建我的DirectShow应用程序，如何配置开发环境?

请参考本书1.3节的相关介绍，或者参考《DirectShow开发指南》一书的5.1节。

20. GraphEdit能显示很多Filter，但其中一部分Filter在SDK文档中并没有说明。请问这些Filter是干什么的?

GraphEdit能够列举出所有在系统中注册的Filter，这些Filter中的一部分是DirectShow本身集成的，其他的可能是第三方软件安装的，也可能是微软公司的相关产品（如Windows Media、NetMeeting等）安装的。另外，一些Filter包装了系统中的编解码器、硬件设备，它们都有自己的名字，看起来也好像增加了很多Filter。

21. 在程序中，如何保证我能构建正确的Filter Graph?

首先你应该在GraphEdit中模拟构建，成功后再在程序中以GraphEdit中构建的Filter Graph为模板调用相关的API构建。

22. 我如何能确认DirectShow已经在指定的机器上安装?

调用如下代码创建Filter Graph Manager组件对象，如果创建成功了，说明DirectShow安装正常。

```
IGraphBuilder *pGraph = NULL;
HRESULT hr = CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_
SERVER,IID_IGraphBuilder, (void **) &pGraph);
```

23. 在不显示Filter的属性页的前提下，如何设置Filter的参数?

你需要知道该Filter上实现的参数设置接口，然后在程序中调用这些接口相关的接口方法。

24. 为什么我无法使用ASF Writer和ASF Reader?

在DirectX 9.0之前，在应用程序或GraphEdit中使用这两个Filter是需要提供证书的。(证书是使用Windows Media Format SDK 7.0必需的，可以从微软公司获得。)在DirectX 9.0中，这两个Filter使用了Windows Media Format SDK 7.1。这个版本的SDK在读取、生成不受DRM (Digital Rights Management)保护的内容时不需要证书。因此，如果是操作不受DRM保护的文件，可以不用证书；但如果要操作受DRM保护的文件，仍然需要证书。另外，Windows Media Format 9 Series SDK发布了这两个Filter的更新的版本。

25. DirectShow能自动通知应用程序当前的播放位置吗?

目前没有这种通知机制。但是应用程序中可以设置一个定时器，不断调用Filter Graph Manager上的IMediaSeeking接口的接口方法GetCurrentPosition来获取当前的播放位置。

26. 我写好了一个Filter，但如何安装它呢?

请参考SDK文档的相关介绍，或《DirectShow开发指南》一书的2.2节。

27. 我能使用GraphEdit来测试我的Filter吗?

GraphEdit能帮你完成一些Filter特性的基本测试，但它并不是Filter健壮性测试的可靠平台。

28. 我怎么能知道一个Filter有多少个Pin?

可以调用Filter上的IBaseFilter接口的接口方法EnumPins进行枚举。

29. Filter运行在系统哪个特权级别上?

Filter运行在系统的Ring 3特权级别上，但一些Filter能控制运行在Ring 0级别上的硬件设备。更多信息可以参考SDK文档，或《DirectShow开发指南》一书的2.10节。

30. 我需要安装一个内核调试器吗?

这取决于你具体的项目。安装Debug版本的DirectX运行时库后，意味着你安装了Debug版本的驱动程序以及其他内核模式的组件。你的程序在运行时，如果在上述这些组件中引

发了一个Debug断言，你的机器会自动重启，除非你已经安装并给你的程序绑定了一个内核调试器。

31. DEFINE_GUID这个宏是怎么工作的？

DEFINE_GUID宏很好地解决了GUID的extern引用问题。例如，假设你的项目中有3个源文件（Src1.cpp、Src2.cpp和Src3.cpp）都需要用到某个GUID，但是GUID只能在整个项目范围内定义一次，而其他地方要用必需声明为extern，使用DEFINE_GUID宏就能方便地解决这个问题。假如在MyGuids.h文件定义了如下GUID：

```
DEFINE_GUID(CLSID_MyObject,
0x00000000, 0x0000, 0x0000, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00);
```

在3个源文件中都包含MyGuids.h文件，但仅在任意一个源文件中、MyGuids.h文件之前包含initguid.h文件，如下：

```
// Src1.cpp
#include <initguid.h>
#include "MyGuids.h"

// Src2.cpp
#include "MyGuids.h"

// Src3.cpp
#include "MyGuids.h"
```

在源文件中，如果没有包含initguid.h，DEFINE_GUID会对GUID创建一个extern引用；如果包含了initguid.h，DEFINE_GUID会被重新定义，以生成一个GUID常量。当你没有在任何源文件中包含initguid.h时，你可能会碰到一个unresolved external symbol（无法识别的外部符号）的连接错误。如果你在两个或两个以上的源文件中包含了initguid.h，你可能会碰到一个redefinition; multiple initialization（重复定义）的编译错误。解决这些问题的方法是，确认只在一个源文件中包含了initguid.h，并且注意避免在头文件中包含initguid.h，因为头文件常常会被包含到各个源文件中去。

注：以下内容来自http://www.gdcl.co.uk。翻译工作得到了该网站的所有人Geraint Davies先生的授权和支持。Geraint Davies先生是DirectShow技术的最初设计者之一，目前他在英国的北威尔士经营一家DirectShow相关技术的咨询公司。感谢他卓越的工作！

32. 为什么设计IAsncReader接口？CPullPin类有什么用？

DirectShow中的大部分流处理在源头上都使用基于请求的模式。也就是说，你告诉上一级Filter（的Pin）哪里开始、哪里结束，然后上一级Filter就会将你请求区段内的数据以一系列Sample的方式传递到你的数据读取线程的Receive函数。这种方式能够胜任大多数回放的应用场合。但也有一些场合，它的效率并不理想，如播放AVI格式的文件。其实，AVI文件格式并不是为顺序播放而设计的（主要是为编辑设计的），因为你需要的数据块经常存放在错误的地方。为了得到正确的数据，你不得不常常参考文件内的索引信息。为了实

现高效的回放，你需要有随机访问文件内容的能力（而不像MPEG一样，顺序地、收到多少数据就分析多少数据）。

为了抽取MPEG Splitter和AVI Splitter中文件读取操作这一公共部分，组成一个独立的Filter，就是现在的File Source Filter（这样做的好处还有，Source可以替换为其他Filter，如File Source URL，用于读取来自网络的数据），需要定义一个接口，使两种数据读取方式（顺序读取和随机读取）都能获得较高的效率。使用这个接口，每个Sample都能够独立请求随机位置的数据。你可以在你的数据读取线程中使用同步方式（SyncRead），也可以使用异步方式（先请求，然后再等待请求的完成）。

CPullPin类提供了IAsncReader这个接口的使用方法。它的设计，使得通过IAsncReader接口读取数据的输入Pin，与工作在一般推模式下的输入Pin（使用IMemInputPin::Receive接收数据）看起来是一样的。如果你的Filter连接到一个调用IMemInputPin::Receive传送数据的输出Pin，那么数据会直接传入你的输入Pin的Receive函数；如果你的Filter连接到一个实现了IAsyncReader接口的输出Pin，CPullPin就会帮助你使用IAsyncReader接口“拉”数据，然后送到你的输入Pin的Receive函数。这样看来，似乎CPullPin的介入，并没有真正地从IAsyncReader接口可以在随机位置读取不同数据的能力上获得多少好处。但是，如果我告诉你，MPEG Splitter使用了CPullPin（同步模式），而AVI Splitter直接使用了IAsyncReader接口，使用它自己的代码来进行非连续数据的异步读取，你吃惊吗？

在同步模式下，File Source Filter只是读文件。当你调用IAsyncReader::SyncReadAligned时，Source Filter并不缓存这个请求，而是直接读取文件数据，然后返回给调用者。此时，CPullPin一般有一个循环过程：得到一个空的Sample，调用同步的数据读取函数，最后将满载数据的Sample传递给你的输入Pin。

在异步模式下，你可以首先调用IAsyncReader::Request提出一个请求，这个请求在File Source Filter中会被放入一个请求队列，并由一个专门的工作线程处理。你随后可以调用IAsyncReader::WaitForNext等待一个请求的完成。因为Win95不支持Overlapped方式的IO，因此我相信File Source Filter在Windows NT下也不会使用Overlapped方式进行文件读取，即使这种方式在Windows NT下能够得到支持。

CPullPin在异步模式下有一个循环过程：将一个请求放入队列，等待一个完成了的请求，如此反复，并且保持总有一个请求放到队列中等待处理。在这一点上，同步和异步两种模式的惟一不同之处在于：在异步模式下，下一个请求总是被立即送到队列中保存，而在同步模式下，下一个请求不会发出，除非你的输入Pin的Receive函数调用返回了。（输入Pin的Receive函数调用，是将Sample一级一级往下传送，直到Renderer Filter的过程。通常这个过程是很快的。）

因此在一个高带宽的应用中，如果你想在你的输入Pin的Receive函数返回之前发出一个请求让Source准备数据，就可以考虑采用异步模式。在其他场合，使用同步模式一样是可以的。如果你想从很多不连续的位置读取数据，你可能要完全抛弃CPullPin，而使用你自己的代码（直接使用IAsyncReader接口）。不过，对于大部分应用来说，CPullPin的同步模式完全能满足你的需要。

另外，IAsyncReader接口对于MPEG Splitter也是很有价值的。因为MPEG Splitter的输入Pin在与File Source的输出Pin连接的过程中，会读取部分数据进行文件格式的检验。如果

是IMemInputPin接口，除非Filter Graph运行起来，否则它是得不到任何数据的！

33. DirectShow中停止（Stopped）、暂停（Paused）、运行（Running）状态之间的区别？

DirectShow Filter Graph有3种可能的状态：停止、暂停和运行。如果一个Graph是停止的，Filter最小地占用资源，并且不进行数据处理。如果Graph运行起来，Filter最大地使用资源进行数据处理，并最终显示或播放。

如果一个Graph处于暂停状态，数据在被处理但是并不（持续）播放。Source Filter将启动并持续将数据推给后面的Transform Filter; Transform Filter处理输入Pin上收到的数据后再往下推，直到Renderer Filter。然后Transform Filter将被阻塞住，因为Renderer Filter并没有释放Sample（或者Renderer Filter的输入Pin的Receive函数并没有返回）；Transform Filter阻塞之后Source Filter自然也会被阻塞住。这样，整个Graph为迅速切换到运行状态做好了准备。

在暂停状态下，Video Renderer会显示第一帧，但是Audio Renderer不会进行声音的播放。因此，有时候如果需要重画视频窗口，可以通过让Filter Graph进入暂停状态然后返回原先状态的方式来实现。

34. 一个典型的Filter Graph有多少个线程？

典型情况下，DirectShow Filter Graph为每个流都使用一个线程。也就是说，Source Filter的输出Pin上有一个线程，负责将数据推给Transform Filter，最终到达Renderer Filter。Transform Filter进行的各种转换或往下的数据传送一般也都发生在这个线程中（即输入Pin的Receive函数中）。一个带有分析/分离功能的Filter或有一个以上输出Pin的Filter，通常每个输出Pin上都会有一个新的线程（除了Source Filter推数据的线程外）。输出Pin上可以使用COutputQueue类来达到这个目的。

除此之外，典型的情况下，还有一个Filter Graph Manager的线程和应用程序线程。

35. g_Templates有什么用？为什么我的程序没有它就无法连接？

Filter是一种COM组件。DirectShow基类库文件Strmbase.lib提供了COM类工厂实现，以简化我们的Filter开发工作。这些类工厂使用了你提供的模板实现，即我们在g_Templates数组中填写的Filter注册信息，包括DLL中实现了几个COM对象，每个对象创建的入口函数等，以及模板的数量（保存到g_cTemplates）。

但是，如果你只是开发使用到了基类库的应用程序，就没有必要定义这个模板：你可以将g_cTemplates设为0，那么g_Templates将不会再被引用。

如果你开发的是一个实现COM组件的DLL，但使用了另外的类工厂实现（如一个ATL项目），此时你的DLL导出的DllGetClassFactory可能就有问题了，因为这个函数在多个地方实现了，包括ATL库、MFC库，还有DirectShow的基类库。此时，你需要确认你想使用的库会首先被找到，否则即使你定义了g_cTemplates和g_Templates，你也不能创建任何COM对象。

36. Filter注册成功了，但发现Filter模块不对，怎么办？

你需要确认你的DLL是从基类库的DllEntryPoint函数进入的（这个函数在dllentry.cpp中实现）。否则，基类库的全局变量g_hInst将不会被初始化为当前模块的句柄，注册也就出了问题。最好的解决方法是，在C/C++运行时（Runtime）初始化中调用DllEntryPoint函数：

```
extern "C" BOOL WINAPI DllEntryPoint(HINSTANCE, ULONG, LPVOID);
BOOL WINAPI DllMain(HANDLE hDllHandle, DWORD dwReason, LPVOID lpReserved)
{
    return DllEntryPoint(reinterpret_cast<HINSTANCE>(hDllHandle),
        dwReason, lpReserved);
}
```

37. 为什么MPEG-1的帧定位这么慢?

DirectX 8（以及8.0a）的MPEG-1 Stream Splitter有一个Bug，使得帧定位（IMediaSeeking使用TIME_FORMAT_FRAMES格式）变得很慢，而且不精确。定位操作需要花费的时间与定位的位置在文件中的偏移量成正比，例如定位到文件的3分钟的位置需要5秒的时间，而定位到文件4分钟的位置可能需要6秒的时间。这个Splitter在任何情况下进行帧定位时，都假设帧率是恒定的。我们可以使用ConvertTimeFormat或者改成基于时间的定位来避开这个Bug。

38. Media Player为什么两次创建我的Filter?

这可能是Windows Media Player 8的一个Bug，而在版本9可能已经被纠正了。如果你注册了一个Source Filter来处理私有协议myproto://parameters…，WMP8可能会把你的Filter创建两次，而且两个实例都会被加入到Filter Graph中，都会被调用IFileSourceFilter::Load。对于这个问题，最好的解决办法是，重写Filter的JoinFilterGraph函数，然后在这个函数中检查是否已经有一个你的Filter的实例存在于当前的Filter Graph中。如果有了，JoinFilterGraph函数就返回一个错误码。

39. 我的应用程序在调用FilterGraph->Release()时阻死了，怎么办?

可能你的进程中同时创建了多个Filter Graph。导致阻死的原因，可能是Filter Graph Manager控制上的死锁（deadlock）。因为在同一进程中的所有Filter Graph会共享一个工作线程。假设现在这个工作线程想要获取Graph A的锁定权，但Graph A的锁定权正被应用程序主线程占据着，于是工作线程阻塞等待。在工作线程阻塞的时候，应用程序主线程还是可以处理窗口消息的，在消息处理的过程中，可能要释放Graph B（仍然在主线程中）。于是Graph B在析构时发送一个消息通知工作线程自己的离去。但是，这时工作线程是阻塞的，根本无法处理这个消息。于是，出现了工作线程等主线程，主线程等工作线程的死锁现象。

你可能会碰到这种麻烦问题：应用程序主线程正在为其他Filter Graph处理事件（如视频窗口的激活和失效），正巧这时候你的Filter Graph要销毁。如果你想在应用程序级别解决这个问题，最简单的方法就是在销毁Filter Graph之前调用InSendMessage判断主线程是否正在处理消息。如果是，暂时不要销毁你的Filter Graph，而使用PostMessage发送一个消息，以延迟你的Filter Graph的销毁。

其他的解决方法也是有的，比如使用无线程的Filter Graph，它的CLSID为

CLSID_FilterGraphNoThread。前提是，这种Filter Graph必须在具有消息循环的线程中创建。

40. 如何使用DirectShow播放实时流？

为了播放实时（网络）流（MPEG压缩格式），你可能要自己写一个Source Filter，以便接收数据并将数据引入到Filter Graph中。这个Filter的基类可能选择CSource，在内部可能要使用套接字（Socket）接收UDP组播数据，然后将数据推下去。

一般来说，Source Filter大都通过IAsyncReader接口来供应数据。这个接口允许数据的随机访问。在你的Source Filter实现IAsyncReader接口是可行的，但不是一定要这样做。你可能更希望让Source Filter工作在推模式，使用IMemInputPin接口向下传送数据。

遗憾的是，DirectShow提供的MPEG-1和MPEG-2 Splitter不是为播放实时流设计的。它们除了分离MPEG流中的音视频数据外，还做两件事：一是将MPEG流中的PTS时间转换成DirectShow的时间戳给Sample打上；二是创建Pin上使用的媒体类型，信息从MPEG的序列头（Sequence Header）中获取。两者都要在Pin的连接过程中，在MPEG文件中进行搜索（并读取数据），这对于实时流显然是不可能的。

对于这个问题，有4种解决方案：

（1）实现一个拉模式的Source Filter。这个Filter在连接之前预接收一定量的MPEG数据。这样，你可以使用现有的DirectShow MPEG Splitter。但是，实际上这种方法实现起来是比较复杂的，工作量可能超过开发一个新的Splitter。

（2）有些“热心人”提供了一套支持播放实时流的MPEG Splitter和Decoder。如果你愿意的话，这可能是最简单的方案。不过，有些Filter的质量并不是很高。

（3）DirectX 8包含了一个叫MPEG-2 Demultiplexer的Filter，它能支持实时流的分离，包括MPEG-2程序流和传输流。但有个问题，就是你必须预先知道数据流的格式，需要分离哪种数据流。在应用程序中对这个Filter进行适当的配置很关键。

（4）你可以开发自己的Splitter。这可能没有你想象中那么困难，因为你的Splitter只需支持实时流的分离，而且不用实现IMediaSeeking接口。你可以研究一下DirectX SDK 6中的mpegparse例子（遗憾的是，这个例子从DirectX SDK 8之后就不再发布了，但仍然可以从相关网站上下载到）。mpegparse有一些Bug，而且只支持拉模式，但它很好地演示了MPEG-1和MPEG-2数据流分离的基本原理。另外，你要开发一个推模式的Source Filter（而不是实现IAsyncReader接口的拉模式Source Filter）。你的Splitter需要在每一次断点之后寻找第一个PTS，以及第一个序列头，作为Sample时间戳转换和Pin上使用的媒体类型。Splitter连接的时候一般使用一个默认的媒体类型，等找到序列头后再动态更新。

B PID扩展插件开发

DirectShow是一个功能强大的开放性应用架构。使用这个架构，我们很容易就能开发出广义的流式数据的处理程序。一般认为，DirectShow除了扩展性良好的架构外，另一块核心的部分就在于它对Filter Graph的控制，即Filter Graph Manager所完成的工作。

微软公司并没有公开Filter Graph Manager的源代码。是否就是说，开发者无法扩展DirectShow控制部分的功能？几乎是这样的，但也不尽然。我们发现，微软公司给开发者留了一道小门：可以通过开发PID（Plug-In Distributor）插件，来有限地增强Filter Graph Manager的功能。

B.1 PID应用原理

PID是一种COM组件。它一般实现了一个标准的Filter Graph Manager不支持的接口，并且被Filter Graph Manager运行时聚合使用。（注：聚合是COM组件功能复用的一种方式。）

当Filter Graph Manager被要求查询一个它本身不支持的接口时，它会到注册表的如下位置查询：

```
HKEY_CLASSES_ROOT\Interface\IID\Distributor
```

其中，IID就是接口的GUID字符串形式。如果待查询的接口可以匹配上述的IID，则Filter Graph Manager创建一个上述注册表位置指定的CLSID（Distributor子键的默认值）的COM组件（这个COM组件就是PID，它实现了当前待查询的接口）。值得注意的是，对于应用程序来说，PID的存在是透明的。PID实现的扩展接口就好像是Filter Graph Manager本身就支持的。

由于PID总是被Filter Graph Manager聚合使用的，PID可以在内部实现时从它的外部聚合者（即Filter Graph Manager）上获得所有的控制接口进行调用，如IFilterGraph::EnumFilters、IFilterGraph::FindFilterByName等。这样，PID实现的自定义接口就可以很方便地增强Filter Graph Manager的功能。

另外，PID插件上还可以可选地实现一个标准的IDistributorNotify接口。因为当Filter Graph Manager聚合一个PID插件时，会查询这个插件上是否支持IDistributorNotify接口。通过这个接口，Filter Graph Manager会通知插件一些Filter Graph的状态改变信息。例如：

- 当Filter Graph状态转换时，会分别调用插件上的IDistributorNotify::Run、IDistributorNotify::Pause、IDistributorNotify::Stop。
- 当Filter Graph被设置了一个同步源（即参考时钟）时，会同时调用插件上的IDistributorNotify::SetSyncSource。
- 当Filter Graph中增加或删除了Filter，或者Pin的连接改变了的时候，会同时调用插件上的IDistributorNotify::NotifyGraphChange。

B.2 PID插件开发

PID插件开发需要注意以下几点：

- 插件必须支持COM的聚合应用。
- 插件必须支持一个标准的Filter Graph Manager不支持的接口。
- 插件实现的自定义接口必须在注册表下适当的位置（上文已提及）注册。
- 插件根据实际需要决定是否实现IDistributorNotify接口。
- 插件内部获得Filter Graph Manager上的控制接口时，不要增加额外的引用计数。否则可能会导致Filter Graph Manager无法销毁。（插件内部可以放心地引用Filter Graph Manager上的各控制接口，因为插件是被聚合的，它的生存周期受Filter Graph Manager控制。插件存活期间，Filter Graph Manager一定存在。）
- 插件必然是被聚合使用的，所以在插件的构造函数中传入的外部聚合者IUnknown指针不能是NULL。PID插件也必然被Filter Graph Manager聚合，可以从外部聚合者上查询是否支持IGraphBuilder接口来确认。

下面，结合实例，我们来介绍在VC环境下PID插件的开发方法。

1. 需求来源

DirectShow的智能连接功能很有用，它能帮助开发者以最简洁的方式来构建Filter Graph（开发者只需编写很少量的代码）。但是，这种方式构建的Filter Graph具有不确定性，即无法预料Filter Graph中到底用了哪些Filter，用到的Filter也未必是我们想要的。既然微软公司提供了完成MPEG1解码的全套Filter，对于播放MPEG1文件，我们希望完全使用微软公司的Filter，希望Filter Graph Manager能够提供这样的一个接口，比如就叫IMPEG1Builder。

2. 开发环境

DirectShow SDK 9.0、VC 6.0、Windows 2000。

3. 实现步骤

（1）用VC生成一个Win32 DLL项目：PidMpeg1，像开发DirectShow Filter一样配置这个项目。

（2）使用系统的GuidGen.exe程序产生两个GUID，分别指定给PID插件和IMPEG1Builder接口，并为IMPEG1Builder接口定义各个接口方法：

```
//
// FltGuids.h
//

#ifndef __H_FltGuids__
#define __H_FltGuids__
// PID插件的CLSID定义
// {595F9D6B-D770-4bd6-8D43-18D0B0751D0D}
DEFINE_GUID(CLSID_PID_MPEG1Builder, 0x595f9d6b,
    0xd770, 0x4bd6, 0x8d, 0x43, 0x18, 0xd0, 0xb0, 0x75, 0x1d, 0xd);

// IMPEG1Builder接口的GUID定义
// {C49B8BB8-5217-4738-B245-FB913F312194}
DEFINE_GUID(IID_IMPEG1Builder, 0xc49b8bb8,
    0x5217, 0x4738, 0xb2, 0x45, 0xfb, 0x91, 0x3f, 0x31, 0x21, 0x94);

#endif // __H_FltGuids__

//
// IMPEG1Builder.h
//

#ifndef __IMPEG1Builder_h__
#define __IMPEG1Builder_h__

#ifdef __cplusplus
extern "C" {
#endif

//----------------------------------------------------------------------
// IMPEG1Builder interface
//----------------------------------------------------------------------
DECLARE_INTERFACE_(IMPEG1Builder, IUnknown)
{
    // 判断一个指定的文件是否是MPEG1格式的文件
    STDMETHOD(IsMPEG1File) (THIS_
        const char * inFilePath
    ) PURE;
    // 为一个MPEG1文件创建播放的Filter Graph
    STDMETHOD(RenderMPEG1File) (THIS_
        const char * inFilePath
```

```
    ) PURE;
};
//--------------------------------------------------------------------

#ifdef __cplusplus
}
#endif

#endif // __IMPEG1Builder_h__
```

（3）定义COM组件类CMPEG1Builder，并让它从CUnknown和IMPEG1Builder派生：

```
// CMPEG1Builder.h
class CMPEG1Builder     : public CUnknown
                        , public IMPEG1Builder
                        , public IDistributorNotify
```

（4）在COM组件类CMPEG1Builder实例化之前，进行外部聚合者的身份验证，即这个外部聚合者不能为NULL，而且必须是Filter Graph Manager：

```
// PidMpeg1.cpp
CUnknown * WINAPI CMPEG1Builder::CreateInstance(LPUNKNOWN lpunk, HRESULT
    *phr)
{
    ASSERT(phr);
    // lpunk不能为NULL，即PID插件必须被聚合使用
    if (!lpunk)
    {
        *phr = VFW_E_NEED_OWNER;
        return NULL;
    }
    // 确认外部聚合者是Filter Graph Manager
    IGraphBuilder * pGraphBuilder = NULL;
    if (FAILED(lpunk->QueryInterface(IID_IGraphBuilder,
        (void**)&pGraphBuilder)))
    {
        *phr = E_FAIL;
        return NULL;
    }
    if (pGraphBuilder)
    {
        pGraphBuilder->Release();
    }
    // 现在可以安全创建一个PID插件对象了
    CUnknown *punk = new CMPEG1Builder(lpunk, phr);
    if (punk == NULL)
    {
        if (phr)
        {
            *phr = E_OUTOFMEMORY;
        }
    }
    return punk;
```

```
}
```

（5）在CMPEG1Builder类中声明IMPEG1Builder接口的各接口方法，并在NonDelegatingQueryInterface函数的实现中暴露这个接口：

```
// CMPEG1Builder.h
class CMPEG1Builder, public IMPEG1Builder
{
    // 其他成员定义（省略）
    // ……

public:
    // --- IMPEG1Builder接口方法声明---
    STDMETHODIMP IsMPEG1File(const char * inFilePath);
    STDMETHODIMP RenderMPEG1File(const char * inFilePath);
};

// CMPEG1Builder.cpp
// 在PID插件上暴露IMPEG1Builder接口
STDMETHODIMP CMPEG1Builder::NonDelegatingQueryInterface(REFIID riid, void
    ** ppv)
{
    CheckPointer(ppv, E_POINTER);

    if (riid == IID_IMPEG1Builder)
    {
        return GetInterface((IMPEG1Builder *) this, ppv);
    }
    else if (riid == IID_IDistributorNotify)
    {
        return GetInterface((IDistributorNotify *) this, ppv);
    }
    else
    {
        return CUnknown::NonDelegatingQueryInterface(riid, ppv);
    }
}
```

（6）实现自定义接口的各个接口方法。首先是IsMPEG1File——判断指定文件是否是MPEG1文件。我们可以采用一个“取巧”的方法：让设定这个文件的File Source Filter与MPEG-1 Stream Splitter试连接，如果连上则说明这是一个MPEG1文件，否则就不是。具体实现如下：

```
// CMPEG1Builder.cpp
STDMETHODIMP CMPEG1Builder::IsMPEG1File(const char * inFilePath)
{
    // 获得Filter Graph Manager的接口
    IGraphBuilder * pBuilder = RetrieveGraphBuilder();
    // 为指定的文件创建一个Source Filter
    IBaseFilter * pSource = NULL;
    WCHAR  wcFilePath[MAX_PATH];
    MultiByteToWideChar(CP_ACP, 0, inFilePath, -1, wcFilePath, MAX_PATH);
```

```
    pBuilder->AddSourceFilter(wcFilePath, L"Source", &pSource);
    pSource->Release();
    // 创建微软公司的MPEG-1 Stream Splitter
    IBaseFilter * pSplitter = NULL;
    CoCreateInstance(CLSID_MPEG1Splitter , NULL, CLSCTX_INPROC_SERVER,
        IID_IBaseFilter, (void **)&pSplitter);
    pBuilder->AddFilter(pSplitter, L"MPEG-1 Splitter");
    pSplitter->Release();
    // 将Source Filter与MPEG-1 Splitter试连接
    IPin * pOut = GetPinByDirection(pSource, PINDIR_OUTPUT);
    IPin * pIn  = GetPinByDirection(pSplitter, PINDIR_INPUT);
    HRESULT hr  = pBuilder->ConnectDirect(pOut, pIn, NULL);
    // 删除Source Filter和MPEG-1 Splitter
    pBuilder->RemoveFilter(pSplitter);
    pBuilder->RemoveFilter(pSource);
    // 如果连接成功，说明这是一个MPEG1文件，否则就不是
    return SUCCEEDED(hr) ? S_OK : S_FALSE;
}
```

另外，RenderMPEG1File函数的实现如下：

```
// CMPEG1Builder.cpp
STDMETHODIMP CMPEG1Builder::RenderMPEG1File(const char * inFilePath)
{
    // 从外部聚合者获得IGraphBuilder接口
    IGraphBuilder * pBuilder = RetrieveGraphBuilder();
    // 为指定的文件创建一个File Source Filter
    IBaseFilter * pSource = NULL;
    WCHAR  wcFilePath[MAX_PATH];
    MultiByteToWideChar(CP_ACP, 0, inFilePath, -1, wcFilePath, MAX_PATH);
    pBuilder->AddSourceFilter(wcFilePath, L"Source", &pSource);
    pSource->Release();
    // 创建微软公司的MPEG-1 Stream Splitter
    IBaseFilter * pSplitter = NULL;
    CoCreateInstance(CLSID_MPEG1Splitter , NULL, CLSCTX_INPROC_SERVER,
        IID_IBaseFilter, (void **)&pSplitter);
    pBuilder->AddFilter(pSplitter, L"MPEG-1 Splitter");
    pSplitter->Release();
    // 创建微软公司的MPEG-1 Video Decoder
    IBaseFilter * pVideoDec = NULL;
    CoCreateInstance(CLSID_CMpegVideoCodec, NULL, CLSCTX_INPROC_SERVER,
        IID_IBaseFilter, (void **)&pVideoDec);
    pBuilder->AddFilter(pVideoDec, L"MPEG-1 Video Decoder");
    pVideoDec->Release();
    // 创建微软公司的MPEG-1 Audio Dedocer
    IBaseFilter * pAudioDec = NULL;
    CoCreateInstance(CLSID_CMpegAudioCodec, NULL, CLSCTX_INPROC_SERVER,
        IID_IBaseFilter, (void **)&pAudioDec);
    pBuilder->AddFilter(pAudioDec, L"MPEG-1 Audio Decoder");
    pAudioDec->Release();
    // 得到File Source Filter的输出Pin和Splitter的输入Pin
    IPin * pOut = GetPinByDirection(pSource, PINDIR_OUTPUT);
    IPin * pIn  = GetPinByDirection(pSplitter, PINDIR_INPUT);
```

```
    // 将File Source与Splitter相连
    HRESULT hr  = pBuilder->ConnectDirect(pOut, pIn, NULL);
    // 得到Splitter的视频部分的输出Pin和Video Decoder的输入Pin
    pOut = GetPinByName(pSplitter, "Video");
    pIn  = GetPinByDirection(pVideoDec, PINDIR_INPUT);
    if (pOut && pIn)
    {
        // 将Splitter输出的视频分支连接到Video Decoder
        hr = pBuilder->ConnectDirect(pOut, pIn, NULL);
    }
    // 得到Splitter的音频部分的输出Pin和Audio Decoder的输入Pin
    pOut = GetPinByName(pSplitter, "Audio");
    pIn  = GetPinByDirection(pAudioDec, PINDIR_INPUT);
    if (pOut && pIn)
    {
        // 将Splitter输出的音频分支连接到Audio Decoder
        hr = pBuilder->ConnectDirect(pOut, pIn, NULL);
    }

    IPin * connected = NULL;
    pIn = GetPinByDirection(pVideoDec, PINDIR_INPUT);
    hr  = pIn->ConnectedTo(&connected);
    // 如果Video Decoder的输入Pin已连上，则将Video Decoder的输出Pin连到Video
    // Renderer，否则删除Video Decoder（说明这个MPEG1文件不含视频内容）
    if (SUCCEEDED(hr) && connected)
    {
        connected->Release();
        hr = pBuilder->Render(GetPinByDirection(pVideoDec,
            PINDIR_OUTPUT));
    }
    else
    {
        pBuilder->RemoveFilter(pVideoDec);
    }
    // 如果Audio Decoder的输入Pin已连上，则将Audio Decoder的输出Pin连到Audio
    // Renderer，否则删除Audio Decoder（说明这个MPEG1文件不含音频内容）
    pIn = GetPinByDirection(pAudioDec, PINDIR_INPUT);
    hr  = pIn->ConnectedTo(&connected);
    if (SUCCEEDED(hr) && connected)
    {
        connected->Release();
        hr = pBuilder->Render(GetPinByDirection(pAudioDec, PINDIR_
            OUTPUT));
    }
    else
    {
        pBuilder->RemoveFilter(pAudioDec);
    }
    return hr;
}
```

（7）完成PID插件的注册和反注册。首先填写COM组件的注册信息：

```
// PidMpeg1.cpp
CFactoryTemplate g_Templates[] =
{
    {
        L"PID MPEG1 Builder",
        &CLSID_PID_MPEG1Builder,
        CMPEG1Builder::CreateInstance,
        NULL,
        NULL
    }
};
int g_cTemplates = sizeof(g_Templates) / sizeof(g_Templates[0]);
```

然后来看几个主要的导出函数的实现:

```
// PidMpeg1.cpp
// DLL的入口函数
extern "C" BOOL WINAPI DllEntryPoint(HINSTANCE, ULONG, LPVOID);
BOOL APIENTRY DllMain(HANDLE hModule,DWORD  dwReason, LPVOID lpReserved)
{
    return DllEntryPoint((HINSTANCE)(hModule), dwReason, lpReserved);
}
// 组件的注册函数
STDAPI DllRegisterServer()
{
    HRESULT hr = NOERROR;
    // 将PID插件上实现的IMPEG1Builder接口注册
    hr = RegisterPIDInterface(IID_IMPEG1Builder, TEXT("IMPEG1Builder"),
        CLSID_PID_MPEG1Builder);
    if (SUCCEEDED(hr))
    {
        // 将PID插件注册成为一个COM组件
        hr = AMovieDllRegisterServer2(TRUE);
    }
    return hr;
}
// 将PID插件上实现的接口注册
HRESULT RegisterPIDInterface(CLSID  ClsidInterface,  LPTSTR strName, CLSID
    ClsidDistributor)
{
    TCHAR       achKey[MAX_PATH];
    TCHAR       achClsidInterface[CHARS_IN_GUID];
    TCHAR       achClsidDistributor[CHARS_IN_GUID];
    OLECHAR     achTemp[CHARS_IN_GUID];
    HKEY        hkey, hsubkey;
    LONG        lReturn;
    HRESULT     hr;
    // 将标识PID插件上实现接口的GUID常量转化为字符串形式
    hr = StringFromGUID2(ClsidInterface, achTemp, CHARS_IN_GUID);
    if (SUCCEEDED(hr))
    {
        wsprintf(achClsidInterface, "%ls", achTemp);
        // 将标识PID插件的GUID常量转化为字符串形式
```

```
        hr = StringFromGUID2(ClsidDistributor, achTemp, CHARS_IN_GUID);
    }
    if (FAILED(hr)) return hr;
    wsprintf(achClsidDistributor, "%ls", achTemp);
    // 生成注册表中的入口字符串
    wsprintf(achKey, "Interface\\%s", achClsidInterface);
    lReturn = RegCreateKey(HKEY_CLASSES_ROOT, achKey, &hkey);
    if (ERROR_SUCCESS == lReturn)
    {
        if (NULL != strName)
        {
            // 给接口设置一个名字
            RegSetValue(hkey, NULL, REG_SZ, strName, lstrlen(strName)+1);
        }
        // 创建一个Distributor子键
        lReturn = RegCreateKey(hkey, "Distributor", &hsubkey);
        if (ERROR_SUCCESS == lReturn)
        {
            // 设置Distributor子键的默认值为PID插件的CLSID
            RegSetValue(hsubkey, NULL, REG_SZ, achClsidDistributor,
                lstrlen(achClsidDistributor)+1);
            RegCloseKey(hsubkey);
        }
        RegCloseKey(hkey);
    }
    return HRESULT_FROM_WIN32(lReturn);
}
// 组件的反注册函数
STDAPI DllUnregisterServer()
{
    HRESULT hr = NOERROR;
    // 将PID插件上实现的IMPEG1Builder接口的注册信息注销
    hr = UnregisterPIDInterface(IID_IMPEG1Builder);
    if (SUCCEEDED(hr))
    {
        // 将PID插件（作为一个COM组件）的注册信息注销
        hr = AMovieDllRegisterServer2(FALSE);
    }
    return hr;
}
// 将PID插件上实现的接口的注册信息注销
HRESULT UnregisterPIDInterface(REFCLSID ClsidInterface)
{
    HKEY    hkey;
    LONG    lReturn;
    HRESULT hr;
    TCHAR   achClsidInterface[CHARS_IN_GUID];
    OLECHAR achTemp[CHARS_IN_GUID];
    // 将标识PID插件上实现接口的GUID常量转化为字符串形式
    hr  = StringFromGUID2(ClsidInterface, achTemp, CHARS_IN_GUID);
    if (FAILED(hr)) return hr;
    wsprintf(achClsidInterface, "%ls", achTemp);
```

```
    // 打开“Interface”子键
    lReturn = RegOpenKeyEx(HKEY_CLASSES_ROOT, TEXT("Interface"),
        0, MAXIMUM_ALLOWED, &hkey);
    if (ERROR_SUCCESS == lReturn)
    {
        // 删除接口的注册信息
        EliminateSubKey(hkey, achClsidInterface);
        RegCloseKey(hkey);
    }
    return HRESULT_FROM_WIN32(lReturn);
}
```

B.3　PID插件的使用

使用VC生成一个基于对话框的MFC程序：PIDTest，并在资源编辑器中生成如图B.1所示的用户界面。

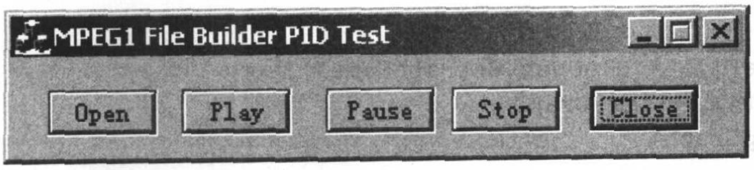


图 B.1　PIDTest 的用户界面

PIDTest程序运行起来，播放一个MPEG1文件的时候，用户界面如图B.2所示。

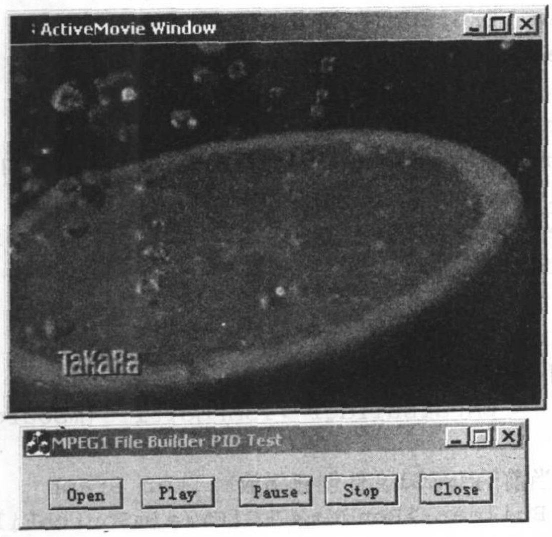


图 B.2　PIDTest 播放文件时的用户界面

PIDTest程序是如何实现的呢？其实，它与开发一般的DirectShow应用程序没有任何区别。使用PID插件，主要就是使用插件上实现的那个接口。首先要声明这个接口的定义：

```
// PIDTestDlg.cpp
```

```
#include "stdafx.h"
#include <streams.h>

#include "IMPEG1Builder.h"
#include <initguid.h>   // 将GUID定义转换成GUID常量
#include "FltGuids.h"
```

然后就是IMPEG1Builder接口的具体使用过程。可以参考Open按钮的响应函数：

```
// PIDTestDlg.cpp
void CPIDTestDlg::OnCmdOpen()
{
    // 弹出标准对话框，让用户选择一个MPEG文件
    CString     strFilter = "MPEG File (*.mpg;*.mpeg)|*.mpg;*.mpeg|";
    CFileDialog dlgOpen(TRUE, NULL, NULL, OFN_HIDEREADONLY, strFilter,
        NULL);
    if (IDOK != dlgOpen.DoModal())
    {
        return;
    }

    mSourceFile = dlgOpen.GetPathName();
    // 重新构建Filter Graph
    if (m_pGraph)
    {
        m_pGraph->Stop();
        delete m_pGraph;
        m_pGraph = NULL;
    }
    m_pGraph = new CDXGraph();
    m_pGraph->Create();

    IMPEG1Builder * pMPG1Builder = NULL;
    // 向Filter Graph Manager查询得到IMPEG1Builder接口
    // IMPEG1Builder接口实际上是由我们的PID插件实现的
    m_pGraph->QueryInterface(IID_IMPEG1Builder, (void**)&pMPG1Builder);
    if (pMPG1Builder)
    {
        pMPG1Builder->Release();
        // 判断是不是MPEG1文件
        if (S_OK == pMPG1Builder->IsMPEG1File(mSourceFile))
        {
            // 为当前的MPEG1文件创建播放Filter Graph
            pMPG1Builder->RenderMPEG1File(mSourceFile);
            // 自动开始播放
            OnCmdPlay();
        }
        else
        {
            AfxMessageBox("This is not a MPEG1 file!");
```

```
            }
        }
    }
```

其中，m_pGraph是主对话框类CPIDTestDlg的成员，它是CDXGraph类的一个实例指针。CDXGraph是一个自定义的Filter Graph Manager的封装类（这个类在本书的前面几章已经被多次使用）。由于源文件以.mpg或.mpeg为扩展名的未必是MPEG1文件（也有可能是MPEG2文件），所以PIDTest首先调用IMPG1Builder::IsMPEG1File进行一下校验。如果不是MPEG1文件，PIDTest会弹出一个消息框进行提示，并不构建该文件的播放链路。

提示：本书配套光盘的SourceCodes\Appendix_B目录下有附录B内容的完整实现：PidMpeg1是插件，PIDTest是插件测试程序。另外，PIDTest程序运行之前，请确认PID插件已经编译成功并完成注册。

C

DES扩展Source（实现字符叠加）

DirectShow主要是为处理顺序的流式数据而设计的，因此在视频的非线性编辑功能上存在“先天不足”。在这样的前提下，DES（DirectShow Editing Services）诞生了。DES实际上是DirectShow API基础上的一层封装，从而弥补了DirecShow的这个缺陷。DES提供的编辑功能包括：视频内容剪辑、音视频效果（Effect）、视频叠加、视频之间的过渡（Transition）、伴音等。唯独没有字符叠加。

字符叠加功能是很有用的，即在一段视频的特定区域内叠加一些说明性的文字。可惜DES没有直接提供这个功能。但是，我们可以通过其他方式来扩展：首先开发一个Source Filter，产生单一背景的字符（图片）流；然后使用DES的色键抠像功能，将字符图片的背景色抠掉，最终叠加到目标视频图像上。

DES对这种扩展Source Filter的开发提出了3点要求：

（1）支持IMediaSeeking接口。

（2）Source Filter输出的数据格式必须是DES能够解码的。

（3）如果要进行自定义属性设置，Source Filter必须实现自动化接口。

前两点容易做到，但是如何来实现第3点呢？因为在实际的字符叠加应用中，有很多属性都需要定制。下面的内容，将对上述3点的实现一一作出介绍。

C.1 自动化基础

大家知道，COM规范具有语言无关性。但实际上，COM接口定义以及COM库提供的API函数中使用的一些数据类型，在一些弱类型的高级语言（如VB、VBA等）中是很难表达的，如复杂的数据结构、指针等。自动化（Automation）作为COM的一个特殊应用，在C++程序与其他一些弱类型语言编写的程序之间搭起了一座桥梁。

自动化的核心是IDispatch接口（每一个自动化对象都必须实现这个接口）。方法（method）和属性（property）是自动化对象的两个基本特性：方法是指自动化对象提供的功能服务，属性是指自动化对象的数据特征。自动化对象的方法和属性都有符号化的名字，客户程序通过名字就可以访问到自动化对象的方法或者属性。另外，类型库（Type Library）的支持是自动化对象的又一个重要特性。

自动化技术的发展与COM技术在VB、VBA中的应用有着直接的关系。然而，因为IDispatch::Invoke函数调用时要进行比较复杂的参数类型检查、转换，以及其他一些类型信息的处理，自动化接口方法的调用同时也是低效率的。于是出现了一种称为双接口（dual interface）的技术。通过实现双接口，自动化对象保留了客户程序通过直接访问虚函数表（vtable）来调用接口方法的高效性。

提示：更多的关于自动化技术的知识，请读者参考介绍COM原理的专著。

C.2 字符叠加基础

Source Filter内如何产生单一背景的字符图片流？即便单一背景的图片很容易产生，但是如何进一步在图片上叠加字符呢？如何能让字符叠加的效率更高一点呢？

这里推荐一种方法：在内存中首先建立一个二色位图，然后在这个位图上画出字符内容。于是，就得到了字符内容的一块点阵信息：如1表示背景，0表示前景字符。创建这个位图的代码参考如下：

```
// CTitleController.cpp
// 参数inDC：内存兼容DC（在此DC中创建二色位图）
HBITMAP CTitleController::ActualCreateTitleDIB(HDC inDC)
{
    // 创建字符位图使用的DIB信息
    // 创建的位图，初始为全黑背景，而在有输出的区域为白底黑字
    // 白底用1表示，黑字用0表示（1和0实际上是下面色彩数组中的索引）
    struct {
        BITMAPINFOHEADER bmiHeader;
        DWORD rgbEntries[2];
    } bmi =
    {
        {
            sizeof(BITMAPINFOHEADER),
```

```
            0,
            0,
            1,
            1,
            BI_RGB,
            0,
            0,
            0
        },
        {
            0x00000000,
            0xFFFFFFFF
        }
    };

    // 字符输出前首先设置字体
    CAutoFont  autoFont;
    if (mDataTitle->IsFontValid())
    {
    //  autoFont.CreateFont("Arial");   // 测试Arial字体
        autoFont.CreateFont(mDataTitle->GetTitleFont());
        autoFont.SelectToDC(inDC);
    }
    char * pTitle = mDataTitle->GetTitleBuffer();
    if (!pTitle)
    {
        return NULL;
    }
    // 获得字符内容的宽、高，以便创建相同大小的二色位图
    GetTextExtentPoint32(inDC, pTitle, strlen(pTitle), &mTitleSize);
    // 别忘了在DIB信息中设置正确的图像尺寸
    bmi.bmiHeader.biHeight = mTitleSize.cy;
    bmi.bmiHeader.biWidth  = mTitleSize.cx;
    // 在指定的DC（实际上是一个内存DC）中创建一个二色位图
    HBITMAP hbm = CreateDIBitmap(inDC, &bmi.bmiHeader, 0, NULL, NULL, 0);
    BOOL   pass = (hbm != NULL);
    if (pass)
    {
        // 将二色位图选入DC中
        HGDIOBJ hobj = SelectObject(inDC, hbm);
        // 在DC中输出字符内容（实际上画在刚选入的二色位图上）
        pass = ExtTextOut(inDC, 0, 0, ETO_OPAQUE | ETO_CLIPPED, NULL,
            pTitle, strlen(pTitle), NULL);
        // 恢复原来DC中的GDI对象
        SelectObject(inDC, hobj);
    }
    // 获取位图数据（相当于点阵信息）
    if (pass)
    {
        SAFE_DELETE_ARRAY(mTitleDib);
        // 注意：图像的宽度是按4字节对齐的
        mTitleDIBStride = ((mTitleSize.cx + 31) >> 3) & ~3;
```

```
        mTitleDib = new BYTE[mTitleDIBStride * mTitleSize.cy];
        memset(mTitleDib, 0, mTitleDIBStride * mTitleSize.cy);
        // 获取位图数据
        LONG lLines = GetDIBits(inDC, hbm, 0, mTitleSize.cy, (PVOID)
            mTitleDib,(BITMAPINFO *)&bmi, DIB_RGB_COLORS);
        pass = (lLines != 0);
    }
    // 如果上述过程中某个步骤失败，则释放使用过的GID资源
    if (!pass && hbm)
    {
        DeleteObject(hbm);
        hbm = NULL;
    }
    return hbm;
}
```

在实际进行字符叠加的时候，将图像帧指定位置的像素与字符点阵的像素位对应：如果字符点阵的像素位值为1（背景色），则保持图像帧对应的像素值不变；如果值为0（前景色），则将图像帧对应的像素值替换为用户设置的字符颜色值。这个叠加算法实现如下：

```
// CTitleController.cpp
// 参数ioFrame：欲叠加字符的图像帧数据
// 参数inXInFrame, inYInFrame：字符在帧内叠加的(X,Y)坐标
void CTitleController::FullTitleOverlayAt(BYTE * ioFrame, long inXInFrame,
long inYInFrame)
{
    // 注意：在GDI中，图像数据采用从下往上的扫描方式存储
    if (mTitleSize.cx > 0 && mTitleSize.cy > 0)
    {
        // 计算实际需要叠加的宽度和高度
        // 如果字符内容太长而超出图像帧的范围，则超出部分不叠加
        long actualOverlayWidth  = min((mDataTitle->GetImageWidth() -
            inXInFrame), mTitleSize.cx);
        long actualOverlayHeight = min((mDataTitle->GetImageHeight() -
            inYInFrame), mTitleSize.cy);

        long  strideInBytes = -mDataTitle->GetImageStride();
        PBYTE pTopLine = ioFrame + mDataTitle->GetImageStride() *
            (mDataTitle->GetImageHeight() - 1);

        // 获取叠加字符的颜色
        DWORD dwTitleColor = mDataTitle->GetTitleColor();
        PBYTE pStartPos = pTopLine + inYInFrame * strideInBytes + inXInFrame
            * mDataTitle->GetBitCount() / 8;

        // 扫描叠加区域对应的点阵信息……
        for (DWORD dwY = 0; dwY < (DWORD)actualOverlayHeight; dwY++)
        {
            PBYTE pbTitle  = mTitleDib + mTitleDIBStride *
                ((DWORD)mTitleSize.cy - dwY - 1);
            for (DWORD dwX = 0; dwX < (DWORD)actualOverlayWidth; dwX++)
            {
```

```
                    // dwX & 7, 取值0 - 7
                    // 0x80 >> (dwX & 7), 取值10000000 - 00000001
                    // dwX >> 3, dwX每增加8时dwX >> 3的值加1
                    // 如果对应的点阵是1（白色，背景色），则保留原图像像素
                    // 如果对应的点阵是0，将原图像对应像素替换成字符颜色
                    if ( !((0x80 >> (dwX & 7)) & pbTitle[dwX >> 3]) )
                    {
                        // 24位位图
                        RGBTRIPLE * prgb = (RGBTRIPLE *) (pStartPos + dwX * 3);
                        prgb->rgbtRed    = GetRValue(dwTitleColor);
                        prgb->rgbtGreen  = GetGValue(dwTitleColor);
                        prgb->rgbtBlue   = GetBValue(dwTitleColor);
                    }
                }
                pStartPos += strideInBytes;
            }
        }
    }
```

C.3 Source Filter的开发

使用一个标准的Win32 DLL项目，配置好Filter的开发环境。Filter的父类可以选择CSource，输出Pin的父类可以选择CSourceStream。

```
// CFilterTitleSource.h
class CFilterTitleSource : public CSource

// CTitleStream.h
class CTitleStream : public CSourceStream
```

1. 在输出 Pin 上实现 IMediaSeeking 接口

DirectShow的基类库中有一个CSourceSeeking类，可以辅助在Source Filter上实现IMediaSeeking接口。这里，可以将CTitleStream类同时从CSourceSeeking类继承：

```
class CTitleStream : public CSourceStream, public CSourceSeeking
```

提示：与Filter一样，Pin其实也是一个COM对象。一个COM对象只能实现一个IUnknown接口，而CSourceStream类和CSourceSeeking类都是从CUnknown类继承下来的（也就都实现了一个IUnknown接口），因而CTitleStream类从CSourceStream和CSourceSeeking双重继承下来是有问题的：运行时会因为引用计数问题而出错。必须修改IUnknown接口的3个接口方法的实现，如下：

```
// CTitleStream.h
STDMETHODIMP QueryInterface(REFIID riid, void **ppv)
{
    if (IID_IMediaSeeking == riid)
    {
```

```
            return CSourceSeeking::QueryInterface(riid, ppv);
        }
        else
        {
            return CSourceStream::QueryInterface(riid,ppv);
        }
    }
    STDMETHODIMP_(ULONG) AddRef()
    {
        return CSourceStream::AddRef();
    }
    STDMETHODIMP_(ULONG) Release()
    {
        return CSourceStream::Release();
    }
```

然后在CTitleStream类中实现CSourceSeeking类的3个纯虚函数：ChangeStart、ChangeStop和ChangeRate。

```
// CTitleStream.h
virtual HRESULT ChangeStart(); // 开始位置改变后被调用
virtual HRESULT ChangeStop();  // 结束位置改变后被调用
virtual HRESULT ChangeRate();  // 播放速率改变后被调用

// CTitleStream.cpp
HRESULT CTitleStream::ChangeStart()
{
    {
        CAutoLock  lock(&mSharedState);
        mSampleTime  = 0;
        if (mController)
        {
            double newPos = m_rtStart * 1.0 / UNITS;
            // 将新时间点告诉控制对象，以产生对应的图像帧
            mController->SeekTo(newPos);
            DbgLog((LOG_TRACE, 0, "Seeking to : %.3f s", newPos));
        }
    }

    // 注意：Seek到一个新位置后需要同步更新数据发送线程
    UpdateFromSeek();
    return NOERROR;
}
// Seek到一个新位置后进行同步更新
void CTitleStream::UpdateFromSeek(void)
{
    // 如果输出Pin上的线程正在活动，则将线程停止后重新启动运行
    // 线程启动后送出的Sample的时间戳将重新从0开始
    if (ThreadExists())
    {
        DeliverBeginFlush();
        Stop();
```

```
            DeliverEndFlush();
            Run();
        }
    }
    HRESULT CTitleStream::ChangeStop()
    {
        return NOERROR;  // 不做处理
    }
    HRESULT CTitleStream::ChangeRate()
    {
        return NOERROR;  // 不做处理
    }
```

最后记得在输出Pin上暴露IMediaSeeking接口：

```
// CTitleStream.cpp
STDMETHODIMP CTitleStream::NonDelegatingQueryInterface(REFIID riid, void
** ppv)
{
    CheckPointer(ppv,E_POINTER);

    if (riid == IID_IMediaSeeking)
    {
        return GetInterface((IMediaSeeking *) this , ppv);
    }
    else
    {
        return CSourceStream::NonDelegatingQueryInterface(riid, ppv);
    }
}
```

2. 在 Filter 上实现自动化（双）接口

自动化接口需要类型库的支持。为了生成类型库，首先要使用对象描述语言（Object Description Language，简称ODL）来描述一个COM组件（包括它实现的接口）。

```
// TitleSource.idl
import "oaidl.idl";
import "ocidl.idl";

[
    // 类型库的GUID
    uuid(D8244BB4-5971-44b8-A150-A4552EC54BBE), // LIBID_TitleSourceType
        Lib
    version(1.0),
    helpstring("Title Source 1.0 Type Library")
]
library TitleSourceTypeLib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    // 下面开始描述ITitleSource接口
```

```
    [
        object,
        uuid(535915D5-CE4A-4781-99DC-D21E2D022F9C), // ITitleSource接口的
            GUID
        helpstring("Title Source Interface"),
        dual
    ]
    // 注意：ITitleSource接口从IDispatch接口继承（所谓的自动化双接口）
    interface ITitleSource : IDispatch
    {
        // 字符的生存期，Duration为属性名
        // get_Duration为读取该属性的方法
        // put_Duration为设置该属性的方法
        [propget, helpstring("Get title duration")]
        HRESULT Duration([out, retval] double * outDuration);
        [propput, helpstring("Set title duration")]
        HRESULT Duration([in] double inDuration);

        // 字符的内容，Title为属性名
        // get_Title为读取该属性的方法
        // put_Title为设置该属性的方法
        [propget, helpstring("Get title")]
        HRESULT Title([out, retval] BSTR outTitle);
        [propput, helpstring("Set title")]
        HRESULT Title([in] BSTR inTitle);

        // 字符的动态效果，Animation为属性名
        // get_Animation为读取该属性的方法
        // put_Animation为设置该属性的方法
        [propget, helpstring("Get title animation type")]
        HRESULT Animation([out, retval] long * outAnimation);
        [propput, helpstring("Set title animation type")]
        HRESULT Animation([in] long inAnimation);

        // 还有更多的字符属性（省略）
        // ……
    }

    // 下面开始描述COM组件（即Source Filter）
    [
        // Source Filter的GUID
        uuid(5AC70BA7-945C-486b-84A4-B2459BAE8209),
            // CLSID_CFilterTitleSource
        helpstring("Title Source Filter class"),
        appobject
    ]
    coclass CFilterTitleSource
    {
        interface ITitleSource;
    }
}
```

然后要给TitleSource.idl文件添加MIDL编译命令，以产生类型库文件、C/C++格式的接口定义文件和GUID定义文件等。方法如下：以“文本”方式打开Filter的项目文件FilterTitleSource.dsp，找到TitleSource.idl项，然后在下面增加“#ADD...”一行，参考如下：

```
# Begin Source File

SOURCE=.\TitleSource.idl
# ADD MTL /tlb ".\TitleSource.tlb" /h "iTitleSource.h" /iid
    "TitleSource_i.c" /Oicf
# End Source File
```

意思是，编译TitleSource.idl文件，生成类型库文件为TitleSource.tlb、接口定义文件为iTitleSource.h、GUID定义文件为TitleSource_i.c。

提示：也可以通过VC的图形界面来设置，步骤是：执行VC菜单命令Project | Settings，在弹出的对话框中，首先在左边选中TitleSource.idl文件，然后在右边单击MIDL标签，随后在Output file name文本框中输入.\TitleSource.tlb、在Output header file name文本框中输入iTitleSource.h、在UUID File文本框中输入TitleSource_i.c。

接着就是ITitleSource和IDispatch的各个接口方法在Filter上的实现。

```
//
// CFilterTitleSource.h
//
// 将CFilterTitleSource类从ITitleSource接口继承
class CFilterTitleSource : public CSource, public ITitleSource

// --- ITitleSource接口方法声明 ---
STDMETHODIMP get_Duration(double * outDuration);
STDMETHODIMP put_Duration(double inDuration);
STDMETHODIMP get_Title(BSTR outTitle);
STDMETHODIMP put_Title(BSTR inTitle);
// 更多接口方法（省略）
// ……
STDMETHODIMP get_Animation(long * outAnimation);
STDMETHODIMP put_Animation(long inAnimation);

// --- IDispatch接口方法声明 ---
STDMETHODIMP GetTypeInfoCount(UINT * pctinfo);
STDMETHODIMP GetTypeInfo(UINT itinfo, LCID lcid, ITypeInfo ** pptinfo);
STDMETHODIMP GetIDsOfNames(REFIID riid, OLECHAR  ** rgszNames,
    UINT cNames, LCID lcid, DISPID * rgdispid);
STDMETHODIMP Invoke(DISPID dispidMember, REFIID riid, LCID lcid, WORD
    wFlags, DISPPARAMS * pdispparams, VARIANT * pvarResult,EXCEPINFO *
    pexcepinfo, UINT * puArgErr);

//
// CFilterTitleSource.cpp
//
```

```
// --- ITitleSource接口方法实现 ---
// TitleStreamPin()函数可以获得输出Pin的指针，该函数实现如下：
// CTitleStream * TitleStreamPin() {return (CTitleStream
// *)m_paStreams[0];};
// mTitleData是数据封装类CDataTitle的实例，在Pin上定义
// CDataTitle类用于保存字符叠加的各个属性
// 得到字符的生存期
STDMETHODIMP CFilterTitleSource::get_Duration(double * outDuration)
{
    *outDuration = TitleStreamPin()->mTitleData.GetDuration();
    return NOERROR;
}
// 设置字符的生存期（仅在输出Pin未连接之前支持更改字符的生存期）
STDMETHODIMP CFilterTitleSource::put_Duration(double inDuration)
{
    if (TitleStreamPin()->IsConnected())
    {
        return E_FAIL;
    }
    TitleStreamPin()->mTitleData.SetDuration(inDuration);
    return NOERROR;
}
// 得到字符内容
STDMETHODIMP CFilterTitleSource::get_Title(BSTR outTitle)
{
    char title[STR_MAX_LENGTH];
    strcpy(title, "");
    TitleStreamPin()->mTitleData.GetTitle(title);
    // 将ANSI字符串转换成宽字符串
    MultiByteToWideChar(CP_ACP, 0, title, lstrlenA(title)+1,
        outTitle, STR_MAX_LENGTH);
    return NOERROR;
}
// 设置字符内容
STDMETHODIMP CFilterTitleSource::put_Title(BSTR inTitle)
{
    char title[STR_MAX_LENGTH];
    // 将宽字符串转换成ANSI字符串
    WideCharToMultiByte(CP_ACP, 0, inTitle, -1, title,
        STR_MAX_LENGTH, NULL, NULL);
    TitleStreamPin()->mTitleData.SetTitle(title, strlen(title));
    return NOERROR;
}
// 得到叠加字符的动画类型
STDMETHODIMP CFilterTitleSource::get_Animation(long * outAnimation)
{
    *outAnimation = TitleStreamPin()->mTitleData.GetTitleAnimation();
    return NOERROR;
}
// 设置叠加字符的动画类型（仅在输出Pin未连接之前支持更改字符的动画类型）
STDMETHODIMP CFilterTitleSource::put_Animation(long inAnimation)
{
```

```
    if (TitleStreamPin()->IsConnected())
    {
        return E_FAIL;
    }
    TitleStreamPin()->mTitleData.SetTitleAnimation((AnimationType)
        inAnimation);
    return NOERROR;
}

// --- IDispatch接口的实现 ---
// IDispatch接口的实现借助于CBasicDispatch类的实例mDispatcher
// CBasicDispatch类主要使用API函数LoadTypeLib或LoadRegTypeLib装载类型库文件
// CBasicDispatch的具体实现参见CBasicDispatch.cpp文件
// 首先在Filter类的构造函数创建CBasicDispatch类的一个实例
CFilterTitleSource::CFilterTitleSource(LPUNKNOWN lpunk, HRESULT *phr) :
CSource(NAME("Title Source"), lpunk, CLSID_CFilterTitleSource)
{
    // 其他初始化代码（省略）
    // ……

    // 获取Filter文件的路径
    // 假设类型库文件与Filter文件放在同一个目录下
    char  achTemp[MAX_PATH];
    GetModuleFileNameA(g_hInst, achTemp, sizeof(achTemp));
    char * backSlash = ::strrchr(achTemp, '\\');
    if (backSlash)
    {
        *backSlash = '\0';
    }
    strcat(achTemp, "\\TitleSource.tlb");
    WCHAR achFileName[MAX_PATH];
    MultiByteToWideChar(CP_ACP, 0, achTemp, lstrlenA(achTemp)+1,
        achFileName, NUMELMS(achFileName));
    // 创建CBasicDispatch类的实例，参数一个是文件路径，一个是类型库的GUID
    mDispatcher = new CBasicDispatch(achFileName,
        LIBID_TitleSourceTypeLib);
}
CFilterTitleSource::~CFilterTitleSource()
{
    // 在Filter退出前释放CBasicDispatch对象
    delete mDispatcher;
}
// --- IDispatch接口的4个接口方法实现 ---
STDMETHODIMP CFilterTitleSource::GetTypeInfoCount(UINT * pctinfo)
{
    // 从类型库中查询信息
    return mDispatcher->GetTypeInfoCount(pctinfo);
}
STDMETHODIMP CFilterTitleSource::GetTypeInfo(UINT itinfo, LCID lcid,
    ITypeInfo ** pptinfo)
{
    // 从类型库获得ITypeInfo接口
```

```
    return mDispatcher->GetTypeInfo(IID_ITitleSource,
        itinfo, lcid, pptinfo);
}
STDMETHODIMP CFilterTitleSource::GetIDsOfNames(REFIID riid,OLECHAR **
    rgszNames,UINT cNames, LCID lcid, DISPID * rgdispid)
{
    // 从类型库中查询信息
    return mDispatcher->GetIDsOfNames(IID_ITitleSource,
        rgszNames, cNames, lcid, rgdispid);
}
STDMETHODIMP CFilterTitleSource::Invoke(DISPID dispidMember, REFIID
    riid,LCID lcid,WORD wFlags, DISPPARAMS * pdispparams,VARIANT *
    pvarResult,EXCEPINFO * pexcepinfo,UINT * puArgErr)
{
    if (IID_NULL != riid)
    {
        return DISP_E_UNKNOWNINTERFACE;
    }
    // 首先从类型库获得ITypeInfo接口
    ITypeInfo * pti;
    HRESULT hr = GetTypeInfo(0, lcid, &pti);
    if (FAILED(hr))
    {
        return hr;
    }
    // 转而调用ITypeInfo接口的Invoke函数
    hr = pti->Invoke((ITitleSource *) this, dispidMember, wFlags,
        pdispparams, pvarResult, pexcepinfo, puArgErr);
    pti->Release();
    return hr;
}

//
// CBasicDispatch.cpp
//
// LoadTypeLib和LoadRegTypeLib两个函数的原型定义
typedef HRESULT (STDAPICALLTYPE *LPLOADTYPELIB)(
    const OLECHAR FAR *szFile,ITypeLib FAR* FAR* pptlib);

typedef HRESULT (STDAPICALLTYPE *LPLOADREGTYPELIB)(REFGUID rguid,
    WORD wVerMajor,
    WORD wVerMinor,
    LCID lcid,
    ITypeLib FAR* FAR* pptlib);

// 装载类型库文件
STDMETHODIMP CBasicDispatch::GetTypeInfo(REFIID riid, UINT itinfo, LCID
    lcid,ITypeInfo ** pptinfo)
{
    CheckPointer(pptinfo,E_POINTER);
    ValidateReadWritePtr(pptinfo,sizeof(ITypeInfo *));
    HRESULT hr;
```

```
*pptinfo = NULL;
// 我们的类型库仅支持一个类型信息接口
if (0 != itinfo)
{
    return TYPE_E_ELEMENTNOTFOUND;
}
if (NULL == pptinfo)
{
    return E_POINTER;
}
// m_pti是ITypeInfo接口对象指针
if (NULL == m_pti)
{
    LPLOADTYPELIB         lpfnLoadTypeLib;
    LPLOADREGTYPELIB      lpfnLoadRegTypeLib;
    ITypeLib              *ptlib;
    HINSTANCE             hInst;

    static const char  szTypeLib[]      = "LoadTypeLib";
    static const char  szRegTypeLib[]   = "LoadRegTypeLib";
    // 装载Ole32Aut.dll
    hInst = LoadOLEAut32();
    if (hInst == NULL)
    {
        DWORD dwError = GetLastError();
        return AmHresultFromWin32(dwError);
    }
    // 在Ole32Aut.dll中查询LoadRegTypeLib函数的入口
    lpfnLoadRegTypeLib = (LPLOADREGTYPELIB)GetProcAddress(hInst,
        szRegTypeLib);
    if (lpfnLoadRegTypeLib == NULL)
    {
        DWORD dwError = GetLastError();
        return AmHresultFromWin32(dwError);
    }

    // 使用LoadRegTypeLib函数装载类型库
    // mTypelibID是类型库的GUID
    hr = (*lpfnLoadRegTypeLib)(mTypelibID, 1, 0, // version 1.0
        lcid, &ptlib);
    // 如果使用LoadRegTypeLib装载类型库失败，则尝试使用LoadTypeLib函数
    if (FAILED(hr))
    {
        // 在Ole32Aut.dll中查询LoadTypeLib函数的入口
        lpfnLoadTypeLib = (LPLOADTYPELIB)GetProcAddress(hInst,
            szTypeLib);
        if (lpfnLoadTypeLib == NULL)
        {
            DWORD dwError = GetLastError();
            return AmHresultFromWin32(dwError);
        }
```

```
            // 使用LoadTypeLib函数装载类型库
            hr = (*lpfnLoadTypeLib)(mTypelibFile, &ptlib);
            if (FAILED(hr))
            {
                return hr;
            }
        }
        // 从类型库获得ITypeInfo接口指针
        hr = ptlib->GetTypeInfoOfGuid(riid, &m_pti);
        ptlib->Release();
        if (FAILED(hr))
        {
            return hr;
        }
    }
    // 输出ITypeInfo接口
    *pptinfo = m_pti;
    // 为输出的ITypeInfo接口增加一个引用计数
    // 注意：使用者在使用完ITypeInfo接口后必须调用一次Release函数
    m_pti->AddRef();
    return S_OK;
}
// 得到类型库中定义的类型信息接口的数量
STDMETHODIMP CBasicDispatch::GetTypeInfoCount(UINT * pctinfo)
{
    CheckPointer(pctinfo,E_POINTER);
    ValidateReadWritePtr(pctinfo,sizeof(UINT *));
    *pctinfo = 1;  // 仅有一个类型信息接口
    return S_OK;
}
// 通过查询类型库，完成名字与内部ID的对应转换
STDMETHODIMP CBasicDispatch::GetIDsOfNames(REFIID riid, OLECHAR  **
    rgszNames,UINT cNames, LCID lcid, DISPID * rgdispid)
{
    ITypeInfo * pti;
    // 从类型库中为riid标识的接口获得一个ITypeInfo接口
    HRESULT hr = GetTypeInfo(riid, 0, lcid, &pti);
    if (SUCCEEDED(hr))
    {
        // 调用ITypeInfo接口的GetIDsOfNames函数实现名字与内部ID的对应转换
        hr = pti->GetIDsOfNames(rgszNames, cNames, rgdispid);
        pti->Release();
    }
    return hr;
}
```

最后别忘了在Filter上暴露ITitleSource和IDispatch两个接口：

```
// CFilterTitleSource.cpp
STDMETHODIMP CFilterTitleSource::NonDelegatingQueryInterface(REFIID riid,
    void ** ppv)
{
    CheckPointer(ppv, E_POINTER);
```

```
    if (riid == IID_IDispatch)
    {
        return GetInterface((IDispatch *)this, ppv);
    }
    else if (riid == IID_ITitleSource)
    {
        return GetInterface((ITitleSource *) this, ppv);
    }
    else
    {
        return CSource::NonDelegatingQueryInterface(riid, ppv);
    }
}
```

3. Filter 的输出媒体类型

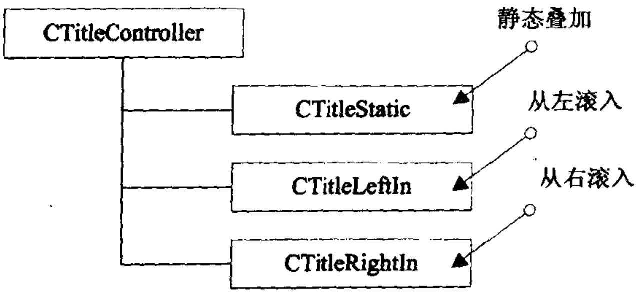
Filter输出的媒体类型在输出Pin的GetMediaType函数中提供。由于RGB是非压缩格式，DES肯定能够支持。所以，让输出Pin提供一种24位位图的媒体类型。

```
// CTitleStream.cpp
HRESULT CTitleStream::GetMediaType(int iPosition, CMediaType *pmt)
{
    CAutoLock  lockit(m_pFilter->pStateLock());
    if (iPosition != 0)   // 仅推荐一种媒体类型
    {
        return E_INVALIDARG;
    }
    // 数据格式使用VIDEOINFO数据结构来描述
    // 首先为格式数据分配一块内存……
    VIDEOINFO * pvi = (VIDEOINFO *) pmt->AllocFormatBuffer(sizeof
        (VIDEOINFO));
    if (NULL == pvi)
    {
        return(E_OUTOFMEMORY);
    }
    ZeroMemory(pvi, sizeof(VIDEOINFO));
    // 填写具体的格式信息
    pvi->AvgTimePerFrame = REFERENCE_TIME(UNITS / mTitleData.GetFrame
        Rate());
    pvi->bmiHeader.biCompression    = BI_RGB;
    pvi->bmiHeader.biBitCount       = 24;
    pvi->bmiHeader.biSize           = sizeof(BITMAPINFOHEADER);
    pvi->bmiHeader.biWidth          = mTitleData.GetImageWidth();
    pvi->bmiHeader.biHeight         = mTitleData.GetImageHeight();
    pvi->bmiHeader.biPlanes         = 1;
    pvi->bmiHeader.biSizeImage      = GetBitmapSize(&pvi->bmiHeader);
    pvi->bmiHeader.biClrImportant   = 0;
    SetRectEmpty(&(pvi->rcSource));
    SetRectEmpty(&(pvi->rcTarget));
    // 设置媒体类型
    pmt->SetType(&MEDIATYPE_Video);
```

```
    pmt->SetFormatType(&FORMAT_VideoInfo);
    pmt->SetTemporalCompression(FALSE);
    // 自动识别媒体子类型
    const GUID SubTypeGUID = GetBitmapSubtype(&pvi->bmiHeader);
    pmt->SetSubtype(&SubTypeGUID);
    pmt->SetSampleSize(pvi->bmiHeader.biSizeImage);
    return NOERROR;
}
```

4. Filter的控制逻辑（实现字符叠加）

Source Filter使用一个数据封装类CDataTitle来保存各种参数设置，使用逻辑控制类CTitleController来控制实际字符的叠加。根据叠加字符的效果（如静态叠加、从左滚入、从右滚入等），可以从CTitleController派生不同的控制类。这些类的结构如图C.1所示：



图C.1 控制类继承结构

CTitleController类的公共接口定义如下：

```
// CTitleController.h
class CTitleController
{
    // 类成员定义（省略）
    // ……
public:
    CTitleController();
    virtual ~CTitleController();

    // 设置参数类对象指针
    void SetDataTitle(CDataTitle * inTitle);
    // 当Seek到新位置后告诉控制类，以便产生新位置对应的图像帧
    void SeekTo(double inCurrentPos);
    // Filter开始运行之前进行初始化，包括字符点阵信息的生成
    virtual HRESULT StartStreaming(void);
    // Filter停止运行时进行一些处理
    virtual HRESULT StopStreaming(void);
    // Filter框架调用该函数实现真正的字符叠加
    virtual HRESULT GetTitleFrame(BYTE * outFrame);

    // 其他接口定义（省略）
    // ……
};
```

GetTitleFrame函数的实现如下：

```
// CTitleController.cpp
HRESULT CTitleController::GetTitleFrame(BYTE * outFrame)
{
    // 叠加前进行一些必要的准备
    BOOL pass = BeforeTitleOverlay(outFrame);
    if (pass)
    {
        // 进行真正的字符叠加
        pass = DoTitleOverlay(outFrame);
    }
    if (pass)
    {
        // 字符叠加完成之后进行一些善后处理
        pass = AfterTitleOverlay();
    }
    return pass ? S_OK : S_FALSE;
}
```

BeforeTitleOverlay函数一般进行进度判断（当字符的生存期过了之后应该让Source Filter停止输出）、单一化图像帧（保证图像在字符叠加前是单纯的背景色）等；DoTitleOverlay函数进行真正的字符叠加，内部使用上文介绍的FullTitleOverlayAt函数，或者另外一个MaskTitleOverlayAt函数（支持滚动字符的叠加）；AfterTitleOverlay函数进行进度累计。以上3个函数都是虚函数，可以供子类根据实际需要重写。这样的设计，使控制类具有良好的扩展性。

C.4 Source Filter的调试

Filter开发完成之后需要进行调试。现成的调试平台为GraphEdit（即SDK目录下的\Bin\DXUtils\graphedt.exe）。配置方法：执行VC菜单命令Project | Settings，在弹出的对话框中，首先在左边选中项目名FilterTitleSource，然后在右边单击Debug标签，再在Executable for debug session文本框中输入GraphEdit程序的完整路径。

将GraphEdit运行起来，插入一个名为DES Ext Source的Filter，然后构建如图C.2所示的Filter Graph。运行它，可以看到如图C.3所示的效果。（如果产生的视频不正常，可以在程序中设置断点进行跟踪、调试。）

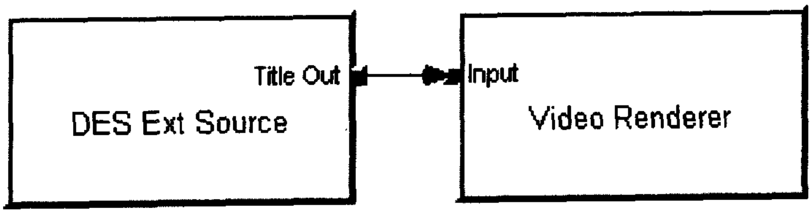
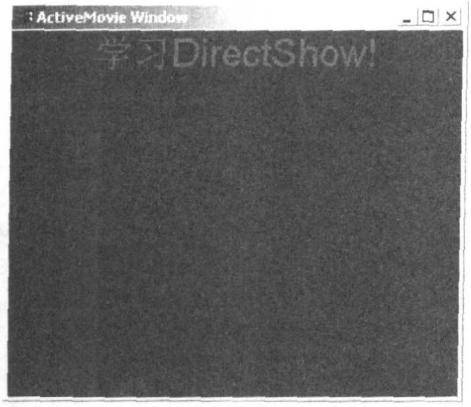


图 C.2　测试 DES Ext Source 的 Filter Graph



图C.3　DES Ext Source产生的单一（蓝）背景字符图片流

提示：使用DES Ext Source Filter之前，请确认该Filter已经被成功注册。另外，DES Ext Source是一个实现了自动化接口的Filter，自动化接口的使用需要类型库的支持。类型库文件TitleSource.tlb必须与Filter文件放在同一个目录下。

C.5　DES应用举例

Source Filter开发完成了，下面就来介绍如何在DES中使用它，以最终实现我们想要的字符叠加功能。大家知道，DES提供了一个色键抠像叠加的Transition组件，它的GUID为CLSID_DxtKey（字符串形式为{C5B19592-145E-11D3-9F04-006008039E37}）。这个组件提供了很多方式的抠像实现，包括根据RGB值、亮度、Alpha值、色调等，很有用！

DES使用XML格式来描述一个视频编辑项目（文件扩展名取为.xtl）。作为演示，我们也来编写一个：使用一个轨道（track）来装载两个SDK自带的媒体文件lake.mpg和highway.avi，在时间线上依次定位为0~4秒和4~10秒；使用另外一个轨道来容纳两个DES Ext Source（即clsid为{5AC70BA7-945C-486b-84A4-B2459BAE8209}的组件），一个在0~3秒内，输出的字符内容为“学习DirectShow!”，另一个在4~8秒内，输出的字符内容为“HQ Tech, Make Technology Easy!”；最后为这两个轨道编排一个过渡，即DES的色键抠像Transition（放在第二个轨道内），抠像的方式为根据RGB值，抠除的颜色值为0x0000ff（蓝色）（因为DES Ext Source产生的字符图片背景色默认是蓝色），作用时间为0~10秒。

```
<!-- Demo_mytitle.xtl -->
<timeline>
  <group type="video" bitdepth="24" width="352" height="288">
    <track>
        <clip src="E:\DXSDK9\Samples\Media\lake.mpg" start="0" stop="4"/>
        <clip src="E:\DXSDK9\Samples\Media\highway.avi" start="4"
            stop="10"/>
    </track>
    <track>
        <clip clsid="{5AC70BA7-945C-486b-84A4-B2459BAE8209}" start="0"
            stop="3">
            <param name="Title"  value="学习DirectShow!"/>
```

```
        <param name="FontName"  value="幼圆"/>
        <param name="FontSize"  value="30"/>
        <param name="Duration"  value="3"/>
        <param name="Animation"  value="1"/>
    </clip>
    <clip clsid="{5AC70BA7-945C-486b-84A4-B2459BAE8209}" start="4"
        stop="8">
        <param name="Title"  value="HQ Tech, Make Technology Easy!"/>
        <param name="TitleColor"  value="0x0000ff00"/>
        <param name="StartX"  value="70"/>
        <param name="StartY"  value="100"/>
    </clip>

    <transition clsid="{C5B19592-145E-11d3-9F04-006008039E37}"
        start="0" stop="10" >
    <param name="KeyType"  value="0"/>
    <param name="RGB"    value="0x0000ff"/>
    <param name="Progress" value="1.0"/>
    </transition>
  </track>

 </group>
</timeline>
```

提示：Demo_mytitle.xtl文件中的编排假设了SDK安装在E:\DXSDK9\目录下。使用该文件演示前，读者应根据SDK实际的安装目录，对上述两个媒体文件的路径进行修正。

Demo_mytitle.xtl文件编辑好后（假设存放在C:\下），可以使用SDK自带的一个例子程序来进行播放。这个例子位于SDK目录下的\Samples\C++\DirectShow\Editing\ XTLTest，编译成功后即可生成xlttest.exe。演示方法：在命令行状态下进入xlttest.exe的所在目录，然后执行命令xtltest.exe C:\Demo_mytitle.xtl，就可以看到如图C.4所示的效果了。

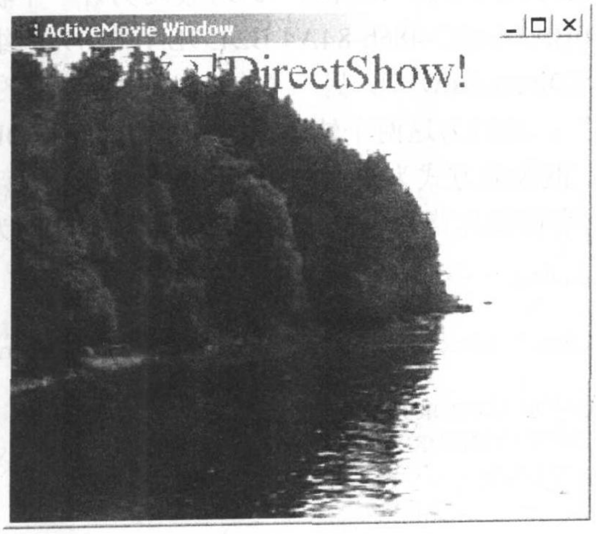


图 C.4　DES 中使用 DES Ext Source 进行字符叠加

提示：本书配套光盘的SourceCodes\Appendix_C目录下有附录C内容的完整实现：FilterTitleSource是DES Ext Source Filter，Demo_mytitle.xtl文件也放在这个项目里。另外，进行DES的扩展演示前，请确认DES Ext Source Filter已经编译成功并完成注册，并且DES Ext Source的类型库文件TitleSource.tlb与Filter文件放在了同一个目录下。

D

让Windows Media Player播放自定义格式文件

由于种种原因，我们可能会生成一些自定义格式的媒体文件（不同于标准的AVI、MPEG等文件格式）。为了回放这些特殊格式的文件，通常需要再开发一个专门的播放器。这可能会令用户感到沮丧，因为他们已经习惯了使用Windows Media Player。也许你的播放器与Windows Media Player风格相似，但是它能拥有Windows Media Player所有的功能吗（以使用户彻底抛弃Windows Media Player）？即使能，那是不是在重复开发，而且工作量还比较大？投入大量的人力、物力、财力开发一个播放器，仅仅是为了比Windows Media Player多支持一种文件格式，值得吗？更何况Windows Media Player还是免费的。

其实，我们可以考虑其他的简单一点的解决方案，如让Windows Media Player自动识别这种自定义的文件格式，进而像播放其他标准文件一样来播放它。能行吗？答案是肯定的。因为Windows Media Player是基于DirectShow架构来播放媒体文件的；只要从DirectShow方面入手，就一定能找到解决问题的办法。

D.1 播放媒体文件的Filter Graph构建过程

DirectShow要播放一个媒体文件，肯定会构建一个Filter Graph。给出了一个源文件的路径，DirectShow首先是要找到一个合适的Source Filter来装载它。一般只需调用IGraphBuilder::RenderFile或者IGraphBuilder::AddSourceFilter就能做到这一点。那么，Filter Graph Manager（的IGraphBuilder接口方法）的实现细节是怎样的呢？了解这些，对于我们

解决问题起着决定性的作用。

首先要分析一下文件路径，因为它可能包含有协议（比如http、htp等）。Filter Graph Manager认为冒号前面的是协议名。例如文件路径为myprotocol://myfile.ext，则myprotocol为协议名。于是，就到注册表的如下位置去查找这个协议：

```
HKEY_CLASSES_ROOT
   <protocol>
      Source Filter = <Source filter CLSID>
      Extensions
         <.ext1> = <Source filter CLSID>
         <.ext2> = <Source filter CLSID>
```

如果在<protocol>位置找到了这个协议，并且注册表的再下一层存在Extensions子键，Filter Graph Manager就再到Extensions子键下进行文件扩展名的匹配。如果匹配成功，就用扩展名指定的GUID字符串作为Source Filter的CLSID；如果不成功，就使用紧跟协议子键下的Source Filter指定的CLSID。如果myprotocol这个协议并没有在注册表中注册，Filter Graph Manager就默认使用一个叫File Source (URL)的Filter（DirectShow集成发布的一个标准的Source Filter）。另外，有两点需要注意：（1）为了区别协议和驱动器盘符（如C:\myfile.ext），Filter Graph Manager不承认单字符的协议；（2）字符串“file:”或“file://”不被认为是协议。

如果文件路径中不包含协议，Filter Graph Manager就分析文件的扩展名，即到注册表的HKEY_CLASSES_ROOT\Media Type\Extensions下去匹配。相应的扩展名子键下指定了Source Filter的CLSID，有的还有Media Type和Subtype（用于指定Source Filter输出Pin上使用的媒体类型）。这部分信息在注册表中的结构参考如下：

```
HKEY_CLASSES_ROOT
Media Type
      Extensions
      <.ext1>
           Source Filter = <Source filter CLSID>
      <.ext2>
           Source Filter = <Source filter CLSID>
           Media Type    = <GUID>
           Subtype       = <GUID>
```

但是，如果指定的文件扩展名没有在上述注册表位置注册，Filter Graph Manager还有最后一招，就是通过文件的校验字节（Check Bytes）去判断。这些校验字节在注册表的

HKEY_CLASSES_ROOT\Media Type\{major type}\{subtype}下注册，注册的信息典型如下：

```
{e436eb83-524f-11ce-9f53-0020af0ba770}
   {7364696D-0000-0010-8000-00AA00389B71}
      0                  "0,4,,52494646,8,4,,524D4944"
      1                  "0,4,,4D546864"
      Source Filter      "{E436EBB5-524F-11CE-9F53-0020AF0BA770}"
```

注意，上面的{e436eb83-524f-11ce-9f53-0020af0ba770}就是{major type}，其实它是MEDIATYPE_Stream的GUID字符串形式。在它下面还定义了一系列的Subtype，有

{e436eb88-524f-11ce-9f53-0020af0ba770}（MEDIASUBTYPE_Avi）、{e436eb84-524f-11ce-9f53-0020af0ba770}（MEDIASUBTYPE_MPEG1System）、MEDIATYPE_Midi（{7364696D-0000-0010-8000-00AA00389B71}）等。在Subtype子键下，还可以看到一些子键，其中编号为0，1，2...的即为定义的校验字节，Source Filter指定一个CLSID。校验字节定义块一般的格式为：offset,cb,mask,val，意思为在文件头偏移offset字节处读取cb个字节的数据，将其与mask做位与操作，结果等于val即表示匹配。如果定义的mask为空，可以不做位与操作；如果offset为负数，表示从文件尾开始计算偏移；如果一个子键有多个定义块，则所有的定义块都匹配才能算这个子键匹配；如果有多个子键用于校验字节的定义，那么任一子键的匹配就表示Source Filter的匹配。如果注册表中定义的所有校验字节都不能匹配，Filter Graph Manager最后就只能默认创建一个名为File Source (Async.)的Filter（DirectShow集成发布的一个标准的Source Filter），输出Pin上的媒体类型使用{MEDIATYPE_Stream, MEDIASUBTYPE_None}。

经过上述这个比较复杂的查找过程之后，Filter Graph Manager最终得到了一个Source Filter的CLSID。创建这个Filter，然后再在Filter上获得IFileSourceFilter接口，通过其接口方法IFileSourceFilter::Load加载将要播放的媒体文件（或者URL）。最后Render这个Source Filter的所有输出Pin，完成Filter Graph的构建。

D.2 两种解决方案

知道了DirectShow播放媒体文件的Filter Graph的构建过程，我们的解决方案也就有了，而且至少有两种。第一种方案，针对自定义格式的媒体文件，开发一个支持拉模式的Parser Filter。这个Filter必须实现一个工作在拉模式的输入Pin（可以参考CPullPin类），完成的主要功能有：在输入Pin的连接过程中，从Source Filter中读取部分数据进行文件格式校验；运行过程中，从Source Filter中读取数据并分析，如果是音视频交互格式的，则要将它们分离，然后从不同的输出Pin输出；实现IMediaSeeking接口；响应Quality Control消息等。另外，必须保证Parser Filter输出的数据在系统中有相应的解码Filter进行解码（如果没有，则另外开发一个）。注意，注册这个Parser Filter（以及相应的解码Filter）时，要让它（们）的Merit值大于MERIT_DO_NOT_USE。

采用第一种方案，我们可以不在注册表中注册额外的信息。在文件回放时，Filter Graph Manager使用默认的File Source (Async.)作为Source Filter，而后面接上我们的Parser Filter（以及相应的解码Filter）。Filter Graph的连接如图D.1所示。

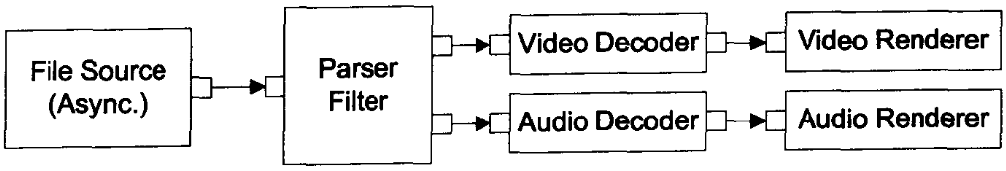


图 D.1　第一种方案下的 Filter Graph 连接

第二种解决方案，可以开发一个推模式的Source Filter。这个Filter类可以从CSource类继承，完成的功能包括：读取文件内容并分析，如果是音视频交互格式，则将它们分离后从不同的输出Pin输出，实现IFileSourceFilter接口，实现IMediaSeeking接口，响应Quality Control消息等。当然，如果这个Source Filter输出的数据不是裸数据，而系统中没有对应的解码Filter，那么另外开发一个解码Filter也是必需的。

采用第二种方案，关键还在于要让Filter Graph Manager在试图播放自定义格式文件的时候，能够自动找到我们的Source Filter。这就需要在Filter注册的时候，同时向注册表中写入一些额外的信息：可以是注册文件扩展名，也可以注册这种文件的校验字节。最终Filter Graph的连接如图D.2所示。

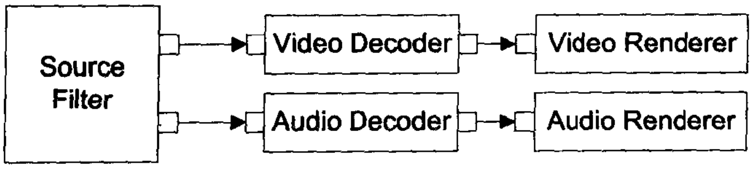


图 D.2 第二种方案下的 Filter Graph 连接

D.3 自定义格式文件的生成

怎么会生成一种自定义格式的文件呢？有一种情况是，在研究某个Filter的输出数据时，需要将这些数据写到一个文件中。这种文件格式不是标准的，现假设它的扩展名为.QQ，并且给它定义一个简单的文件格式：文件头的开始两个字节以QQ作为标识，随后跟4个字节用于指示格式数据的长度，然后跟格式数据（使用VIDEOINFOHEADER数据结构来描述），最后一部分就是图像数据（一帧紧跟着一帧），如图D.3所示。

Q	Q	n	VIDEOINFOHEADER	图像数据
2字节		4字节	n字节	

图 D.3 QQ 文件的文件格式

生成QQ文件的Filter Graph如图D.4所示。

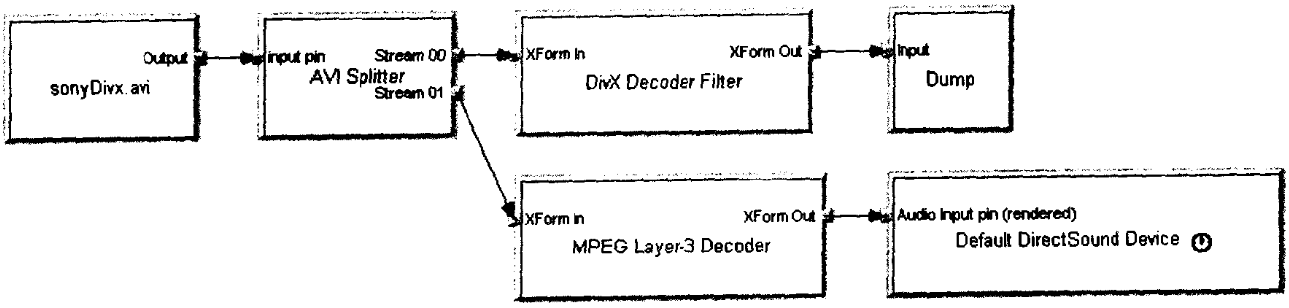


图 D.4 生成 QQ 文件的 Filter Graph 连接

图D.4中用到了一个Dump Filter，它是由SDK的一个Filter例子（这个例子位于SDK目录的\Samples\C++\DirectShow\Filters\Dump下）改写过来的。改写的工作包括：

（1）修改输入Pin的CheckMediaType函数，让它仅接受一些常见的视频媒体类型。

```
// Dump.cpp
HRESULT CDumpInputPin::CheckMediaType(const CMediaType * inMt)
{
    if (inMt->majortype == MEDIATYPE_Video &&
        inMt->formattype == FORMAT_VideoInfo)
    {
        if (inMt->subtype == MEDIASUBTYPE_RGB32 ||
            inMt->subtype == MEDIASUBTYPE_RGB24 ||
            inMt->subtype == MEDIASUBTYPE_RGB555 ||
            inMt->subtype == MEDIASUBTYPE_RGB565 ||
            inMt->subtype == MEDIASUBTYPE_RGB8 ||
            inMt->subtype == MEDIASUBTYPE_YUY2 ||
            inMt->subtype == MEDIASUBTYPE_UYVY ||
            inMt->subtype == MEDIASUBTYPE_YUYV ||
            inMt->subtype == MEDIASUBTYPE_YVYU)
        {
            return S_OK;
        }
    }

    return E_FAIL;
}
```

（2）修改输入Pin的Receive函数，将接收到的Sample数据写入文件。

```
// Dump.cpp
STDMETHODIMP CDumpInputPin::Receive(IMediaSample *pSample)
{
    CheckPointer(pSample,E_POINTER);
    CAutoLock lock(m_pReceiveLock);
    WriteQQFile(pSample);
    return NOERROR;
}

#include <stdio.h>
void CDumpInputPin::WriteQQFile(IMediaSample * inSample)
{
    static long counter = 0;
    static FILE * fp    = NULL;

    // 如果接收的是第一个Sample，则打开文件，并写入文件头信息
    if (counter == 0)
    {
        // 生成的文件为C:\Dump.QQ
        fp = fopen("C:\\Dump.QQ", "wb");
        // 首先写入两个字符：QQ，作为文件标识
        char buf[] = "QQ";
        fwrite(buf, sizeof(char), 2, fp);
```

```
        // 然后写入格式数据块的大小
        // 格式数据块从输入Pin连接用的媒体类型中获得
        VIDEOINFOHEADER * info = (VIDEOINFOHEADER *) m_mt.pbFormat;
        long size = sizeof(*info);
        fwrite(&size, 1, 4, fp);
        // 然后再写入格式数据块
        fwrite(info, 1, sizeof(*info), fp);
    }
    // 将100帧图像数据写入到QQ文件中
    // 一个Sample携带的数据就是一帧图像数据
    if (++counter <= 100 && fp)
    {
        PBYTE pData = NULL;
        inSample->GetPointer(&pData);
        if (pData)
        {
            fwrite(pData, 1, inSample->GetActualDataLength(), fp);
        }
    }
    else if (fp)
    {
        // 写完100帧图像后关闭文件
        fclose(fp);
        fp = NULL;
    }
}
```

D.4 Source Filter的开发

以第二种方案为例，我们来开发一个支持QQ文件播放的Source Filter。

提示：至于第一种方案，读者可以自行实现，方法类似。与第二种方案相比，第一种方案主要在数据获取的方式上有所不同，即要在输入Pin上实现从Source Filter拉数据的功能。具体实现时，可以借助于CPullPin类。

1. Filter 的总体结构

该Filter仅有一个输出Pin。Filter的注册名为QQ Source。Filter的父类为CSource，Pin的父类为CSourceStream。

```
// CFilterQQSource.h
class CFilterQQSource : public CSource

// CQQOutPin.h
class CQQOutPin : public CSourceStream
```

Pin的实例在Filter的构造函数中生成：

```
// CFilterQQSource.cpp
```

```
CFilterQQSource::CFilterQQSource(LPUNKNOWN lpunk, HRESULT *phr) :
CSource(NAME("QQ Source"), lpunk, CLSID_QQSource)
{
    CQQOutPin * outStream = new CQQOutPin(phr, this, L"Output");
    if (outStream == NULL)
    {
        *phr = E_OUTOFMEMORY;
    }
}
```

并且要重写Filter类的GetPinCount和GetPin两个函数，以完成对Filter框架的描述。

```
// CFilterQQSource.cpp
// 得到Filter拥有的Pin的数量
int CFilterQQSource::GetPinCount()
{
    return 1;  // 该Filter仅有1个Pin
}
// 依次得到Filter上的所有Pin对象指针
CBasePin * CFilterQQSource::GetPin(int n)
{
    if (n == 0)
    {
        return m_paStreams[0];
    }
    return NULL;
}
```

因为CSourceStream类封装了一个数据发送线程，会不断调用Pin类上的函数FillBuffer进行Sample内容的填写，然后发送给下一级Filter，所以对源文件内容的读取、定位等操作均放在Pin类中实现。

```
// CQQOutPin.cpp
HRESULT CQQOutPin::FillBuffer(IMediaSample * pSample)
{
    CAutoLock  lock(&mSharedState);
    // 获取Sample中存储数据的内存地址
    BYTE * pData = NULL;
    pSample->GetPointer(&pData);
    // 从文件中读取一帧放到Sample中
    long bytes = fread(pData, 1, mFrameSize, m_fp);
    if (bytes < mFrameSize)
    {
        // 如果所有帧都读完，则返回S_FALSE以结束数据发送线程
        return S_FALSE;
    }

    // 设置Sample的属性，包括时间戳、有效数据长度等
    REFERENCE_TIME start = mLastSampleTime;
    mLastSampleTime += mSampleDuration;
    REFERENCE_TIME stop  = mLastSampleTime;
    pSample->SetActualDataLength(mFrameSize);
    pSample->SetTime(&start, &stop);
```

```
        // 因为Sample中的数据格式是非压缩的，
        // 所以任何一个Sample都可以成为同步点
        pSample->SetSyncPoint(TRUE);
        return S_OK;
}
// 每次（从停止状态）开始播放时，将文件指针定位到第一帧的位置
HRESULT CQQOutPin::Active(void)
{
        if (m_fp)
        {
            CAutoLock   lock(&mSharedState);
            // 定位到文件中存放第一帧的位置（去掉文件头数据！）
            fseek(m_fp, mHeaderSize, SEEK_SET);
        }
        return CSourceStream::Active();
}
```

2. IFileSourceFilter接口的实现

IFileSourceFilter是标准的用于装载源文件的接口，它有两个接口方法：Load和GetCurFile。

```
// CFilterQQSource.h
class CFilterQQSource : public CSource, public IFileSourceFilter
{
public:
    // 其他函数声明（省略）
    // ……

    STDMETHODIMP Load(LPCOLESTR pszFileName, const AM_MEDIA_TYPE *pmt);
    STDMETHODIMP GetCurFile(LPOLESTR *ppszFileName, AM_MEDIA_TYPE *pmt);

};

//
// CFilterQQSource.cpp
//
// OutPin()函数用于获取输出Pin对象的指针
// 它的实现为：CQQOutPin * OutPin() {return (CQQOutPin *)m_paStreams[0];};
STDMETHODIMP CFilterQQSource::Load(LPCOLESTR pszFileName, const
    AM_MEDIA_TYPE *pmt)
{
    // 将源文件设置给输出Pin
    char szAnsi[MAX_PATH];
    WideCharToMultiByte(CP_ACP, 0, pszFileName, -1, szAnsi, MAX_PATH, NULL,
        NULL);
    if (OutPin()->SetFileSource(szAnsi))
    {
        return NOERROR;
    }
    return E_FAIL;
}
STDMETHODIMP CFilterQQSource::GetCurFile(LPOLESTR *ppszFileName,
    AM_MEDIA_TYPE *pmt)
```

```
{
    // 得到当前Source Filter装载的源文件
    char szAnsi[MAX_PATH];
    OutPin()->GetFileSource(szAnsi);
    DWORD n = sizeof(WCHAR) * (1 + strlen(szAnsi));
    // 分配内存（外部调用者负责释放）
    *ppszFileName = (LPOLESTR) CoTaskMemAlloc( n );
    if (*ppszFileName != NULL)
    {
        WCHAR   szwFile[MAX_PATH];
        MultiByteToWideChar(CP_ACP, 0, szAnsi, -1, szwFile, MAX_PATH);
        CopyMemory(*ppszFileName, szwFile, n);
    }
    return NOERROR;
}

//
// CQQOutPin.cpp
//
BOOL CQQOutPin::SetFileSource(const char * inFile)
{
    CAutoLock  lock(&mSharedState);
    // 保存文件路径
    strcpy(mFilePath, inFile);
    if (m_fp)
    {
        fclose(m_fp);
        m_fp = NULL;
    }
    // 将源文件以“二进制读取”方式打开
    m_fp = fopen(mFilePath, "rb");
    mHeaderSize = 0;
    // 检查文件头标识是否为两个字母QQ
    char  buf[10];
    fread(buf, 1, 2, m_fp);
    buf[2] = '\0';
    if (strcmp(buf, "QQ") != 0)
    {
        return FALSE;
    }
    mHeaderSize += 2;
    // 读出格式数据块，输出Pin将使用它来组成媒体类型，以便与下一级Filter连接
    long size = 0;
    fread(&size, 1, 4, m_fp);
    mHeaderSize += 4;
    BYTE * p = new BYTE[size];
    fread(p, 1, size, m_fp);
    mHeaderSize += size;
    // 设置媒体类型的主类型、格式类型
    mPreferred.SetType(&MEDIATYPE_Video);
    mPreferred.SetFormatType(&FORMAT_VideoInfo);
    mPreferred.SetTemporalCompression(FALSE);
    // 获取媒体类型的子类型
    VIDEOINFOHEADER * pvi = (VIDEOINFOHEADER *) p;
    const GUID subtype = GetBitmapSubtype(&pvi->bmiHeader);
```

```
    mPreferred.SetSubtype(&subtype);
    // 计算图像帧的大小、帧率等属性
    mFrameSize = pvi->bmiHeader.biWidth * pvi->bmiHeader.biHeight
        * pvi->bmiHeader.biBitCount / 8;
    pvi->bmiHeader.biSizeImage = mFrameSize;
    // 设置媒体类型的格式数据块
    mPreferred.SetFormat(p, size);
    // 将文件指针定位到文件末尾，以便得到文件的大小
    fseek(m_fp, 0, SEEK_END);
    long pos = ftell(m_fp);
    // 计算该文件包含的总帧数
    mTotalFrames = (pos - mHeaderSize) / mFrameSize;
    // 将文件指针定位到第一帧的位置，以便开始运行时读取
    fseek(m_fp, mHeaderSize, SEEK_SET);
    if (pvi->AvgTimePerFrame > 0)
    {
        mSampleDuration = pvi->AvgTimePerFrame;
    }
    else
    {
        // 默认为25fps
        mSampleDuration = 400000;
    }
    m_rtStop = m_rtDuration = mTotalFrames * mSampleDuration;
    delete[] p;
    return TRUE;
}
void CQQOutPin::GetFileSource(char * outFile)
{
    strcpy(outFile, mFilePath);
}
```

IFileSourceFilter的接口方法实现完成之后，记得在Filter上暴露这个接口：

```
// CFilterQQSource.cpp
STDMETHODIMP CFilterQQSource::NonDelegatingQueryInterface(REFIID riid,
    void ** ppv)
{
    CheckPointer(ppv, E_POINTER);

    if (riid == IID_IFileSourceFilter)
    {
        return GetInterface((IFileSourceFilter *) this, ppv);
    }
    else
    {
        return CSource::NonDelegatingQueryInterface(riid, ppv);
    }
}
```

3. IMediaSeeking 接口的实现

如果想要让Windows Media Player在播放QQ文件的时候能够显示出播放进度，那么Source Filter（的输出Pin）上必须实现IMediaSeeking接口。方法是：首先将输出Pin类从CSourceSeeking类继承，然后实现CSourceSeeking类的3个纯虚函数ChangeStart、ChangeStop

和ChangeRate。

```
// CQQOutPin.h
class CQQOutPin : public CSourceStream, public CSourceSeeking
{
public:
    // 其他函数声明（省略）
    // ……

    // CSourceSeeking类的3个必须实现的纯虚函数
    virtual HRESULT ChangeStart();
    virtual HRESULT ChangeStop();
    virtual HRESULT ChangeRate();
};

// CQQOutPin.cpp
// 开始位置改变后被调用的函数
HRESULT CQQOutPin::ChangeStart()
{
    {
        CAutoLock  lock(&mSharedState);
        mLastSampleTime = 0;
        // 将新的开始时间转换成最近的帧边界时间
        m_rtStart = m_rtStart / mSampleDuration * mSampleDuration;
        long pos  = mTotalFrames * mFrameSize * m_rtStart / m_rtDuration;
        pos += mHeaderSize;
        // 将文件指针定位到一个新的位置
        fseek(m_fp, pos, SEEK_SET);

        DbgLog((LOG_TRACE, 0, "Seeking to : %10d", m_rtStart));
    }
    // 重新启动数据发送线程
    UpdateFromSeek();
    return NOERROR;
}
void CQQOutPin::UpdateFromSeek(void)
{
    // 如果数据线程存活，则将其停止后重新启动
    if (ThreadExists())
    {
        DeliverBeginFlush();
        Stop();
        DeliverEndFlush();
        Run();
    }
}
// 当结束位置改变时被调用，现不做处理
HRESULT CQQOutPin::ChangeStop()
{
    return NOERROR;
}
// 当播放速率改变时被调用，现不做处理
```

```
HRESULT CQQOutPin::ChangeRate()
{
    return NOERROR;
}
```

然后记得在输出Pin上暴露IMediaSeeking接口：

```
// CQQOutPin.cpp
STDMETHODIMP CQQOutPin::NonDelegatingQueryInterface(REFIID riid, void **
    ppv)
{
    CheckPointer(ppv,E_POINTER);

    if (riid == IID_IMediaSeeking)
    {
        return GetInterface((IMediaSeeking *) this , ppv);
    }
    else
    {
        return CSourceStream::NonDelegatingQueryInterface(riid, ppv);
    }
}
```

4. Quality Control 消息响应

因为这是一个推模式的Source Filter，数据在一个独立的发送线程中压迫式地往下传送。于是，Sample达到Renderer的时刻，要么太早，要么太晚。不管是太早还是太晚，Video Renderer都会发出Quality Control消息。如果Sample早到了，Video Renderer会将发送线程阻塞住，直到Filter Graph的流时间到了Sample的开始时间。如果Sample晚到了，问题可能严重一点，因为久而久之的累积效应会造成音视频播放的不同步。因此，Source Filter最好对表示晚到的Quality Control消息作出处理（如加快Sample的发送速度或者进行跳帧）。

对Quality Control消息的处理，也就是要重写输出Pin上的Notify函数：

```
// CQQOutPin.cpp
STDMETHODIMP CQQOutPin::Notify(IBaseFilter * pSender, Quality q)
{
    if (q.Late > 0) // Sample到达Video Renderer的时间晚了
    {
        CAutoLock  lock(&mSharedState);
        // 计算当前正确的流时间
        mLastSampleTime += q.Late;
        // 指向下一帧的时间
        mLastSampleTime += mSampleDuration;
        // 将新的开始时间转换成最近的帧边界时间
        mLastSampleTime  = mLastSampleTime / mSampleDuration *
            mSampleDuration;
        // 将文件指针定位到一个新的位置
        long pos = mTotalFrames * mFrameSize * mLastSampleTime /
            m_rtDuration;
        pos += mHeaderSize;
        fseek(m_fp, pos, SEEK_SET);
```

```
            // 输出调试跟踪信息
            DbgLog((LOG_TRACE, 0, "Quality control seeking to : %10d",
                mLastSampleTime));
        }
        else
        {
            // Sample到达Video Renderer的时间早了（不做处理）
        }
        return NOERROR;
    }
```

5. 自定义接口及属性页的实现

Source Filter实现了IFileSourceFilter接口后会自动生成一个属性页，用于显示源文件的路径。（在GraphEdit中插入一个Source Filter，右击该Filter，从弹出的菜单中选择Filter Properties命令后就可以看到属性页。）但是，在这个属性页上不能更改源文件。如果要播放其他文件，除非重新插入一个Source Filter。

否则，可以另外增加一个属性页。在这之前，先来介绍一下一个自定义接口IQQSource的实现。首先是接口定义：

```
//
// IQQSource.h
//

#ifndef __H_IQQSource__
#define __H_IQQSource__

#ifdef __cplusplus
extern "C" {
#endif

//----------------------------------------------------------------------
// IQQSource的接口函数定义
//----------------------------------------------------------------------
DECLARE_INTERFACE_(IQQSource, IUnknown)
{
    // 询问Filter是否可以改变源文件
    STDMETHOD(CanChangeSource) (THIS) PURE;
};
//----------------------------------------------------------------------

#ifdef __cplusplus
}
#endif

#endif // __H_IQQSource__
```

然后是接口方法的实现：

```
// CFilterQQSource.h
class CFilterQQSource : public CSource, public IFileSourceFilter, public
    IQQSource
{
public:
```

```
    // 其他函数声明（省略）
    // ……
    // --- IQQSource接口方法声明 ---
    STDMETHODIMP CanChangeSource(void);
};

// CFilterQQSource.cpp
// --- IQQSource接口方法实现 ---
STDMETHODIMP CFilterQQSource::CanChangeSource(void)
{
    // 仅在Filter Graph处于停止状态，并且Source Filter的输出Pin没有连接的情况下
    // 允许改变源文件
    if (IsStopped() && OutPin()->IsConnected() == FALSE)
    {
        return S_OK;
    }
    return S_FALSE;
}
// 暴露IQQSource接口
STDMETHODIMP CFilterQQSource::NonDelegatingQueryInterface(REFIID riid,
    void ** ppv)
{
    CheckPointer(ppv, E_POINTER);

    if (riid == IID_IFileSourceFilter)
    {
        return GetInterface((IFileSourceFilter *) this, ppv);
    }
    else if (riid == IID_IQQSource)
    {
        return GetInterface((IQQSource *) this, ppv);
    }
    else
    {
        return CSource::NonDelegatingQueryInterface(riid, ppv);
    }
}
```

属性页的实现，首先要在VC的资源编辑器中编辑如图D.5所示的一个对话框。

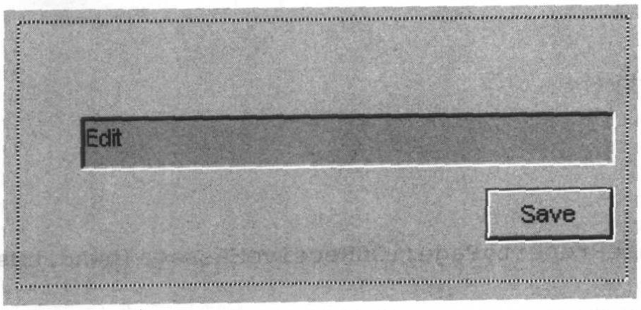


图 D.5　Source Filter 的属性页界面

然后编写一个属性页类来关联这个对话框。这个属性页类从CBasePropertyPage类继承。

```
// CQQSourceProp.h
```

```
class CQQSourceProp : public CBasePropertyPage

// CQQSourceProp.cpp
CQQSourceProp::CQQSourceProp(LPUNKNOWN pUnk, HRESULT *phr) :
CBasePropertyPage(NAME("QQ Prop Page"),pUnk,IDD_DLG_PROP,IDS_INFO)
{
    ASSERT(phr);
}
```

具体实现时，CQQSourceProp类需要实现父类的OnReceiveMessage、OnConnect、OnDisconnect、OnActivate、OnApplyChanges等函数。

```
// CQQSourceProp.cpp
// 属性页窗口的消息处理
BOOL CQQSourceProp::OnReceiveMessage(HWND hwnd, UINT uMsg,WPARAM wParam,
    LPARAM lParam)
{
    switch (uMsg)
    {
    case WM_INITDIALOG:
    {
        // 获取各个控件的窗口句柄
        mEditFile   = GetDlgItem(hwnd, IDC_EDIT_FILE_SOURCE);
        mButtonSave = GetDlgItem(hwnd, IDC_BUTTON_SAVE);
        break;
    }

    case WM_COMMAND:
    {
        // 当用户单击Save按钮时，保存源文件路径到Filter对象中
        if (HIWORD(wParam) == BN_CLICKED)
        {
            switch (LOWORD(wParam))
            {
            case IDC_BUTTON_SAVE:
                EnterFilePath();
                break;
            }
        }
        break;
        SetDirty();
    }

    }

    return CBasePropertyPage::OnReceiveMessage(hwnd,uMsg,wParam,lParam);
}
// 获取Filter上的接口：IQQSource和IFileSourceFilter
HRESULT CQQSourceProp::OnConnect(IUnknown *pUnknown)
{
    HRESULT hr = pUnknown->QueryInterface(IID_IQQSource, (void **)
        &mIQQSource);
```

```
    hr = pUnknown->QueryInterface(IID_IFileSourceFilter, (void **)
        &mIFile);
    if (FAILED(hr))
    {
        return E_NOINTERFACE;
    }
    ASSERT(mIQQSource);
    ASSERT(mIFile);

    return NOERROR;
}
// 释放属性页类中使用的所有接口
HRESULT CQQSourceProp::OnDisconnect()
{
    if (mIQQSource == NULL)
        return E_UNEXPECTED;
    mIQQSource->Release();
    mIQQSource = NULL;

    if (mIFile)
    {
        mIFile->Release();
        mIFile = NULL;
    }
    return NOERROR;
}
// 将属性页显示出来之前，更新各控件的状态
HRESULT CQQSourceProp::OnActivate()
{
    // 从Filter取得当前的源文件
    ReflectFilePath();
    // 如果当前Filter的状态允许改变源文件，则将各控件激活，否则禁止使用各控件
    if (CanChangeSource())
    {
        EnableWindow(mEditFile, TRUE);
        EnableWindow(mButtonSave, TRUE);
    }
    else
    {
        EnableWindow(mEditFile, FALSE);
        EnableWindow(mButtonSave, FALSE);
    }
    return NOERROR;
}
// 当属性页的Apply按钮按下时，设置源文件到Filter中
HRESULT CQQSourceProp::OnApplyChanges()
{
    // 判断当前Filter是否允许更改源文件
    if (CanChangeSource())
    {
        EnterFilePath();
    }
```

```
        return NOERROR;
    }
    // 判断当前Filter是否允许更改源文件
    BOOL CQQSourceProp::CanChangeSource(void)
    {
        if (mIQQSource)
        {
            return (mIQQSource->CanChangeSource() == S_OK);
        }
        return FALSE;
    }
    // 从Filter中获取当前的源文件，然后在属性页相应的控件中显示
    void CQQSourceProp::ReflectFilePath(void)
    {
        if (mIFile)
        {
            CMediaType mt;
            LPOLESTR oldFileName = NULL;
            if (SUCCEEDED(mIFile->GetCurFile(&oldFileName, &mt)))
            {
                char szAnsi[MAX_PATH];
                WideCharToMultiByte(CP_ACP, 0, oldFileName, -1,
                    szAnsi, MAX_PATH, NULL, NULL);
                SetWindowText(mEditFile, szAnsi);
                CoTaskMemFree(oldFileName);
            }
            else
            {
                SetWindowText(mEditFile, "");
            }
        }
    }
    // 设置新的源文件到Filter中
    void CQQSourceProp::EnterFilePath(void)
    {
        if (mIFile)
        {
            char szFile[MAX_PATH];
            if (GetWindowText(mEditFile, szFile, MAX_PATH))
            {
                WCHAR  szwFile[MAX_PATH];
                MultiByteToWideChar(CP_ACP, 0, szFile, -1, szwFile, MAX_PATH);
                mIFile->Load(szwFile, NULL);
            }
        }
    }
```

需要注意的是，属性页也是一种COM组件，需要提供组件对象的创建函数CreateInstance。

```
// CQQSourceProp.cpp
CUnknown * WINAPI CQQSourceProp::CreateInstance(LPUNKNOWN lpunk, HRESULT
```

```
    *phr)
{
    CUnknown *punk = new CQQSourceProp(lpunk, phr);
    if (punk == NULL)
    {
        *phr = E_OUTOFMEMORY;
    }
    return punk;
}
```

属性页准备好之后，最后还需要在Filter上实现ISpecifyPropertyPages接口：

```
// CFilterQQSource.h
class CFilterQQSource : public CSource, public ISpecifyPropertyPages,
    public IFileSourceFilter , public IQQSource
{
public:
    // 其他函数声明（省略）
    // ……
    // --- ISpecifyPropertyPages接口方法声明 ---
    STDMETHODIMP GetPages(CAUUID *pPages);
};

// CFilterQQSource.cpp
// --- ISpecifyPropertyPages接口方法实现 ---
STDMETHODIMP CFilterQQSource::GetPages(CAUUID *pPages)
{
    pPages->cElems = 1;
    pPages->pElems = (GUID *) CoTaskMemAlloc(sizeof(GUID));
    if (pPages->pElems == NULL)
    {
        return E_OUTOFMEMORY;
    }
    *(pPages->pElems) = CLSID_QQSourceProp;
    return NOERROR;
}
// 暴露ISpecifyPropertyPages接口
STDMETHODIMP CFilterQQSource::NonDelegatingQueryInterface(REFIID riid,
    void ** ppv)
{
    CheckPointer(ppv, E_POINTER);

    if (riid == IID_IFileSourceFilter)
    {
        return GetInterface((IFileSourceFilter *) this, ppv);
    }
    else if (riid == IID_IQQSource)
    {
        return GetInterface((IQQSource *) this, ppv);
    }
    else if (riid == IID_ISpecifyPropertyPages)
    {
        return GetInterface((ISpecifyPropertyPages *) this, ppv);
    }
    else
```

```
        {
            return CSource::NonDelegatingQueryInterface(riid, ppv);
        }
    }
```

以及属性页的模板信息填写：

```
// CFilterQQSource.cpp
CFactoryTemplate g_Templates[] =
{
    // Filter的模板信息
    {
        L"QQ Source",
        &CLSID_QQSource,
        CFilterQQSource::CreateInstance,
        NULL,
        &sudFilter
    },
    // 属性页的模板信息
    {
        L"QQ Source Property Page",
        &CLSID_QQSourceProp,
        CQQSourceProp::CreateInstance
    }
};
int g_cTemplates = sizeof(g_Templates) / sizeof(g_Templates[0]);
    // 模板数量
```

6. Filter 的注册

这是最后一步，也是最关键的一步。主要看DllRegisterServer和DllUnregisterServer两个函数的实现。

```
// CFilterQQSource.cpp
STDAPI DllRegisterServer()  // Filter注册函数
{
    // 在注册表中注册.QQ文件扩展名
    DWORD   dwDisposition;
    DWORD   dwReserved = 0;
    HKEY    hTempKey   = (HKEY)0;
    TCHAR   szKey[]    = "Media Type\\Extensions\\.QQ";
    TCHAR   szValue[]  = "Source Filter";
    TCHAR   szData[100];
    WCHAR * pClsid = NULL;
    // 将Filter的CLSID转换成字符串形式
    StringFromCLSID(CLSID_QQSource, &pClsid);
    WideCharToMultiByte(CP_ACP, 0, pClsid, -1, szData, 100, NULL, NULL);
    CoTaskMemFree(pClsid);
    DWORD   dwBufferLength = lstrlen(szData) * sizeof(TCHAR);
    // 使用注册表API函数操作注册表
    if (ERROR_SUCCESS == ::RegCreateKeyEx(HKEY_CLASSES_ROOT, szKey,
        dwReserved, (LPTSTR)0, REG_OPTION_NON_VOLATILE, KEY_SET_VALUE, 0,
        &hTempKey, &dwDisposition))
    {
        dwBufferLength += sizeof(TCHAR);
        ::RegSetValueEx(hTempKey, (LPTSTR)szValue, dwReserved, REG_SZ,
```

```
            (LPBYTE)szData, dwBufferLength);
        ::RegCloseKey(hTempKey);
    }
    return AMovieDllRegisterServer2( TRUE );
}
STDAPI DllUnregisterServer()  // Filter反注册函数
{
    // 删除.QQ文件扩展名在注册表中的注册
    HKEY    hKey = NULL;
    DWORD   dw   = 0;
    TCHAR   szKey[] = "Media Type\\Extensions";
    if (ERROR_SUCCESS == ::RegCreateKeyEx(HKEY_CLASSES_ROOT, szKey, 0L,
        NULL, REG_OPTION_VOLATILE, KEY_ALL_ACCESS, NULL, &hKey, &dw))
    {
        EliminateSubKey(hKey, ".QQ");
        ::RegCloseKey(hKey);
    }
    return AMovieDllRegisterServer2( FALSE );
}
```

D.5 体验QQ文件的播放

在Source Filter开发完成，并且成功注册到系统的前提下，可以首先使用GraphEdit来播放QQ文件。方法是：运行GraphEdit，然后将一个QQ文件拖放进来或者执行GraphEdit的菜单命令File | Render Media File来选中一个待播放的QQ文件。如果Filter Graph自动构建起来，如图D.6一样，则说明我们的Source Filter生效了，QQ文件能够被自动识别了。

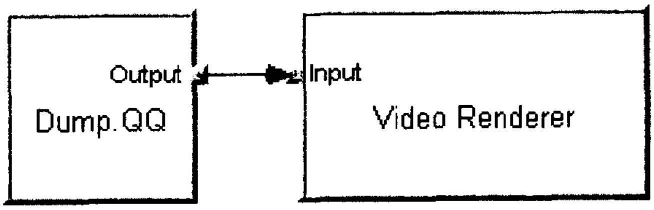


图 D.6 播放 QQ 文件的 Filter Graph 连接

QQ文件在GraphEdit中播放正常后，可以使用Windows Media Player来测试。结果正是我们想要的！

提示：本书配套光盘的SourceCodes\Appendix_D目录下有附录D内容的完整实现：Dump是生成QQ文件的Filter，FilterQQSource是播放QQ文件的Filter。使用Windows Media Player直接播放QQ文件之前，请确认QQ Source Filter已经编译成功并完成注册。

Document generated by Anna’s Archive around 2023-2024 as part of the DuXiu collection (https://annas-blog.org/duxiu-exclusive.html).

Images have been losslessly embedded. Information about the original file can be found in PDF attachments. Some stats (more in the PDF attachments):
{
  "filename": "XzExMjcxOTc3LnppcA==",
  "filename_decoded": "_11271977.zip",
  "filesize": 42447405,
  "md5": "53d6abc73206684be11755fc4dcc60d1",
  "header_md5": "21474542fdcb0e588968841abf5198cf",
  "sha1": "338b71f029d0f74318d11906989ecd07c5768209",
  "sha256": "73d23230fabcb9fcc7678d74a00ef79ed2366ae9b93cd12aa75290d94d606d66",
  "crc32": 305898004,
  "zip_password": "wcpfxk&*^TDwcpfxk",
  "uncompressed_size": 45079999,
  "pdg_dir_name": "",
  "pdg_main_pages_found": 471,
  "pdg_main_pages_max": 471,
  "total_pages": 480,
  "total_pixels": 3180806684,
  "pdf_generation_missing_pages": false
}